LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-449

A SIGNAL PROCESSING LANGUAGE FOR COARSE GRAIN DATAFLOW MULTIPROCESSORS

Janice S. Onanian

June 1989

545 TECHNOLOGY SQUARE, CAMBR

SECURITY CLASSIFICATION OF THIS PAGE

4

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
MIT/LCS/TR-449

5. MONITORING ORGANIZATION REPORT NUMBER(S)
N00014-84-K-0099

6a. NAME OF PERFORMING ORGANIZATION
MIT Laboratory for Computer Science

6b. OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION
Office of Naval Research/Department of Navy

6c. ADDRESS *(City, State, and ZIP Code)*
545 Technology Square
Cambridge, MA 02139

7b. ADDRESS *(City, State, and ZIP Code)*
Information Systems Program
Arlington, VA 22217

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
DARPA/DOD

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*
1400 Wilson Boulevard
Arlington, VA 22217

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
A Signal Processing Language for Coarse Grain Dataflow Multiprocessor

DTIC
ELECTE
OCT 30 1989
S E D

12. PERSONAL AUTHOR(S)
Onanian, J.S.

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
1989 June

15. PAGE COUNT
126

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Coarse-grain Parallelism, Partitioning of Programs, Dataflow Graphs, Data-routing Operators, Basic Objects, Streams, Signals

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This thesis presents a language and graph representation designed to aid in the partitioning of large signal processing applications into tasks to run on a multiprocessor. The language *PGL* compiles directly into a program graph upon which optimizations are performed to find the optimal task configuration. The optimal task configuration is one in which computation-to-communication is minimized while throughput is maximized. The language and graph are designed to express coarse-grain parallelism so that large communication delays and task overhead do not outweigh the speedup achieved by exploiting the parallelism in the application. The dataflow model underlying the graph representation is based on a distributed, loosely-coupled multiprocessor concept which supports tagged dataflow at the operating system level.

The approach utilized in the graph and language is to insert specific, data-routing operators into the dataflow graph defined by the application. These operators shape and route data through the application and express all the coarse-grain parallelism which is

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL
Judy Little, Publications Coordinator

22b. TELEPHONE *(Include Area Code)*
(617) 253-5894

22c. OFFICE SYMBOL

83 APR edition may be used until exhausted.
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆U.S. Government Printing Office: 1985-507-047

Unclassified

exploitable in the model. Data routing operators are parameterized to reflect the degree of parallelism, i.e., the number of parallel tasks in a partition of the application. The graph constructs also contain sufficient information to describe fully the task structure of the application. Given the characteristics of the partition, the programmer can define a standard by which the "goodness" of the partition is evaluated. Given this standard, algorithms to perform the evaluation can be developed. The ultimate goal is the development of algorithms to perform the entire partitioning process automatically.

The problem domain considered in this work is limited to signal processing applications. The set of data-routing operators defined in this thesis is tailored specifically to the characteristics of these applications. By limiting the set of operators, the program graph maintains readability and the language maintains clarity of use. Given the methodology outlined above, accomodating higher levels of generality involves expanding the set of operators and graph constructs. This work is intended to demonstrate the basic program graph methodology and the lanuage $\mathcal{PGL}$. Future work may involve widening applicability of the language, designing the algorithms for the partitioning process and implementation of the current design.

A Signal Processing Language for Coarse Grain Dataflow Multiprocessors

Janice S. Onanian

MIT / LCS / TR-449
June 1989

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099. The author has been supported in part by Raytheon Company, Equipment Division, in Sudbury, Massachusetts.

This report was originally published as the author's master's thesis.

A Signal Processing Language for Coarse Grain Dataflow Multiprocessors

Janice S. Onanian

Technical Report MIT / LCS / TR-449
June 1989

*MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139*

Abstract

This ~~thesis~~ presents a language and graph representation designed to aid in the partitioning of large signal processing applications into tasks to run on a multiprocessor. ~~The~~ language $\mathcal{PGL}$ compiles directly into a program graph upon which optimizations are performed to find the optimal task configuration. The optimal task configuration is one in which computation-to-communication is minimized while throughput is maximized. The language and graph are designed to express coarse-grain parallelism so that large communication delays and task overhead do not outweigh the speedup achieved by exploiting the parallelism in the application. The dataflow model underlying the graph representation is based on a distributed, loosely-coupled multiprocessor concept which supports tagged dataflow at the operating system level.

The approach utilized in the graph and language is to insert specific, data-routing operators into the dataflow graph defined by the application. These operators shape and route data through the application and express all the coarse-grain parallelism which is exploitable in the model. Data routing operators are parameterized to reflect the degree of parallelism, *i.e.*, the number of parallel tasks in a partition of the application. The graph constructs also contain sufficient information to describe fully the task structure of the application. Given the characteristics of the partition, the programmer can define a standard by which the "goodness" of the partition is evaluated. Given this standard, algorithms to perform the evaluation can be developed. The ultimate goal is the development of algorithms to perform the entire partitioning process automatically.

The problem domain considered in this work is limited to signal processing applications. The set of data-routing operators defined in this thesis is tailored specifically to the characteristics of these applications. By limiting the set of operators, the program graph maintains readability and the language maintains clarity of use. Given the methodology outlined above, accommodating higher levels of generality involves expanding the set of operators and graph constructs. This work is intended to demonstrate the basic program graph methodology and the language $\mathcal{PGL}$. Future work may involve widening applicability of the language, designing the algorithms for the partitioning process and implementation of the current design.

Key Words and Phrases: Coarse-grain Parallelism, Partitioning of Programs, Dataflow Graphs, Data-routing Operators, Basic Objects, Streams, Signals.

Acknowledgements

This thesis was completed only with the support and guidance of Professor Arvind. I am deeply grateful for his wisdom, encouragement and confidence in my ability.

I would also like to thank Dan Dechant for his support and guidance in directing the work and his patience and energy in seeing it completed. I am also grateful to Dr. Frank Horrigan and Mr. Robert Soli of Raytheon Company, whose insights were extremely helpful. In addition, I would like to acknowledge Mr. Kevin O'Toole and the VI-A program for providing the opportunity to begin the work and the support needed for its timely completion.

For words of advice and encouragement, my thanks to Greg Papadopoulos, David Culler and Madhu Sharma. Thanks also to Natalie Tarbet, for her effort and thoroughness in proof-reading the final copy.

And finally, the greatest thanks to Mom, Dad and Rachel, without whom, none of this would have been possible.

Contents

List of Figures

Chapter 1

Introduction

Partitioning of large problems into multiple tasks for concurrent execution on multiprocessor machines has been a focus of parallel processing research since the development of such machines. This research has focused on three areas of parallel programming: development of new algorithms, development of new languages, and development of new compilers for existing languages. The first is concerned with determining the data dependencies in a given calculation; the second is concerned with expression of the parallelism in a program. The third is concerned with the difficulties in revealing the parallelism in a program written in a non-parallel language, *i.e.*, a language in which parallelism is not readily expressed.

In designing a language for parallel programming, a major goal is to have as much of the parallelism in a program as possible be derived from the information explicitly or implicitly supplied in the language. The programmer should not have to denote parallelizable sections of code, nor should he have to concern himself with the actual mechanisms for task instantiation, task initialization and termination or inter-task communication [1].

Ideally, transformation of the program into parallel tasks would be done *dynamically*, *i.e.*, at run time. This is the method employed in such machines as MIT's Tagged-Token Dataflow Architecture [2], in which each instruction is a separate task such that very *fine-grain* parallelism is exploited. In most coarse-grain architectures, the task structure of an application is *statically* determined, *i.e.*, at compile time (although assignment of tasks to processors may be dynamic).

The difficulty in dividing a program into parallel tasks has led to languages containing constructs which explicitly denote task assignments. An example of this is the language Ada [6], which contains a special task object type in which the programmer specifies entry points and task interactions. Other languages have been developed similarly but for specific architectures;

an example of this is the TASK language for the CM* multiprocessor [12]. In almost all cases, the partitioning of the problem and the assignment of code blocks to processes is done by the programmer.

There are many high-level languages which express parallelism in a machine-independent manner, but these are generally designed for a much finer grain of parallelism than discussed in this work [8, 13, 11]. Partitioning of data into small pieces for coarse-grain dataflow in multiprocessors has been carried out largely by hand, with throughput, memory and computation to communication ratios calculated by the system designer [24]. Compilers for these problems have mainly taken existing programs in traditional languages and analyzed data dependencies within this context [23]. The level of automation in these compilers cannot yet support full partitioning of problems into an optimal task configuration.

1.1 Advanced On-Board Signal Processor

This thesis is the result of work done at Raytheon's Equipment Division on the Advanced On-Board Signal Processor (AOSP) project. The AOSP is a distributed array of loosely-coupled programmable processors interconnected by a local area network consisting of a packet-based high-speed bus. The system is completely decentralized, with each processor having its own clocks and local memory and with distributed arbitration for the buses. System availability is achieved with fault-recovery techniques designed to assure autonomy. The operating system does not assume any particular number of nodes or interconnect topology and supports transparent communication of messages between tasks. Network topology is highly connected, so that a constant communication latency is a reasonable assumption [9, 15].

Applications are partitioned and mapped onto the network resources in the form of a generalized pipeline with processor interactions synchronized locally by dataflow techniques, *i.e.*, a task is invoked when its inputs arrive (although periodic tasks are supported). Use of the network resources is pre-planned, *i.e.*, tasks are allocated statically, eliminating the need for real-time contention resolution.

1.2 Research Goal

Given this process-oriented, message passing model, the specific goal of this thesis is the development of a high-level language and intermediate representation for large signal processing applications in which:

- coarse-grain parallelism is readily expressed, and
- the optimal partition of the application into tasks can be determined automatically or semi-automatically.

Granularity must be large enough so that individual tasks utilize a significant enough portion of a node's resources to justify the separation of processing into an independent task. Individual task processing latency must be large enough to compensate for task switching overhead and communications processing overhead.

The approach taken in this work to solving the partitioning problem is to use a dataflow representation of an application in which some of the constructs in the representation are parameterized to direct the degree of partitionability. The idea is that the programmer will progr‸m the application in the language described in Chapter 3, called $\mathcal{PGL}$ (for Program Graph Language). The $\mathcal{PGL}$ program will then be transformed into the intermediate representation, called the program graph. The program graph is parameterized; by supplying values for the parameters, the programmer specifies a task partition for the application.

The above approach required the development of a new semantics for the program graph as well as the development of a new class of graph constructs. The program graph has a methodology and a semantics which distinguish it from current representation techniques, including those techniques which are graph-based. Current graph representations have one of two different types of semantics. One is "function-based", with nodes in the graph representing functions or operators which operate on data. The other is "task-based", with nodes in the graph representing tasks in the application job. The former views an application *mathematically*, or in terms of the algorithms and data structures involved. The latter views an application *operationally*, or in terms of the actual processing on a real machine. The program graph in this work is an attempt to bridge the gap between the two representations, the idea being that this is a good way to program that may make decomposing an application easier and more efficient. Later chapters explain the specific motivations for the approach and describe in detail the actual semantics.

One possible scenario for optimizing the partition is as follows: given the task partition, the information in the graph can be used to determine how good the partition is. Since the graph consists of well-defined structures, evaluation can be performed algorithmically as well as manually. If the partition isn't "good" enough, new parameter values can be generated and the process repeats until a satisfactory partition is found. The partition found by this method is not necessarily the optimal one, however, it is good enough by some user-defined criterion based on computational latency, desired throughput, communication overhead and memory requirements.

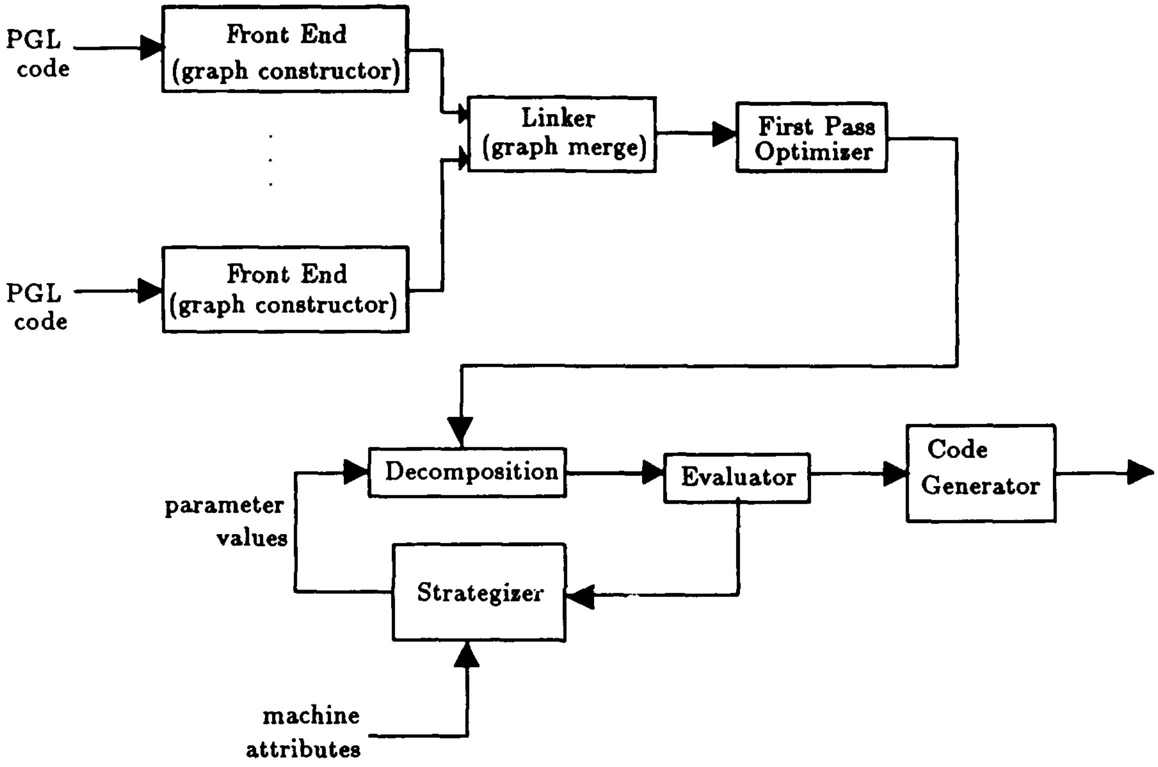


Figure 1.1: Compiler Model

The above optimization process is depicted in Figure 1.1. The figure depicts one possible model for the compiler/optimizer, which consists of several modules interacting to perform the desired analysis. The front end to the compiler is a graph constructor in which a module of $\mathcal{PGL}$ code is transformed into the graph representation. The output of each graph constructor is sent to a linker which merges all module graphs into a complete application graph. The first pass optimizer performs optimizations on the resulting graph structure not related to decomposition. Decomposition refers to the process of supplying values for the graph parameters.

The second phase of the compiler involves decomposition of the application. This phase of the compiler process is beyond the scope of this work. This thesis is concerned only with finding a language and representation and the appropriate transformation rules. The main design goal is to make the subsequent development of evaluation and optimization algorithms feasible and desirable. Consequently, the precise definition of optimality is left for future work; this thesis is concerned only with identification of the information necessary to determine the optimal partition and capturing of this information in a workable representation.

1.3 Problem Limitations

In order to reduce the partitioning problem to a solvable level, the application domain was limited to a special class of signal processing systems. The processing needs of these systems are characterized by a pipelined, functional structure and regular, well-defined data types. The applications which fit this model exhibit very regular patterns of access to large data structures and process large data in sections. A large, target-identifying radar application will be presented which exhibits these characteristics; other applications will be referenced with less detail, and only their salient features discussed.

Optimal partitions of these applications will meet throughput and resource usage requirements as defined by the application and the architecture, respectively.

1.4 Organization

This thesis is organized into six chapters. The first chapter presented the problem to be solved and gave a brief description of current work in this area. The second chapter derives the language and program graph structure from the application structure and dataflow model and describes the requirements of a language and graph representation with the stated design goals. The third chapter presents the high-level language $\mathcal{PGL}$. The fourth chapter presents the program graph constructs derived from the requirements described in the second chapter. The fifth chapter presents the transformation rules used in the graph constructor.

The sixth chapter concludes the work, with outlined suggestions for a possible evaluator/strategizer compiler module. The appendices present a proto-typical application based on the processing needs of a space-based radar and demonstrate the use of the language and graph on an actual application. The appendices contain detailed descriptions of the radar application and portions of the program graph and code for functions within the application.

Chapter 2

Deriving Language Abstractions

The design of the language was motivated by the structure of the applications considered and the level at which dataflow principles are applied. To derive the necessary constructs, several applications were examined for parallelism. One application in particular was chosen for detailed analysis and testing; this target application is a surveillance space-based radar (SBR) system which identifies targets against a background of noise and clutter and extracts relative position and velocity data using fast Fourier transform (FFT) and pulse compression techniques [5]. The details of the radar application are presented in Appendix A.

The next several sections outline and generalize the types of parallelism found in the radar application. The high-level model of an application system and the dataflow model of execution are defined. Language criteria are established based on the material presented and general ideas about modern programming languages.

2.1 High-Level System Model

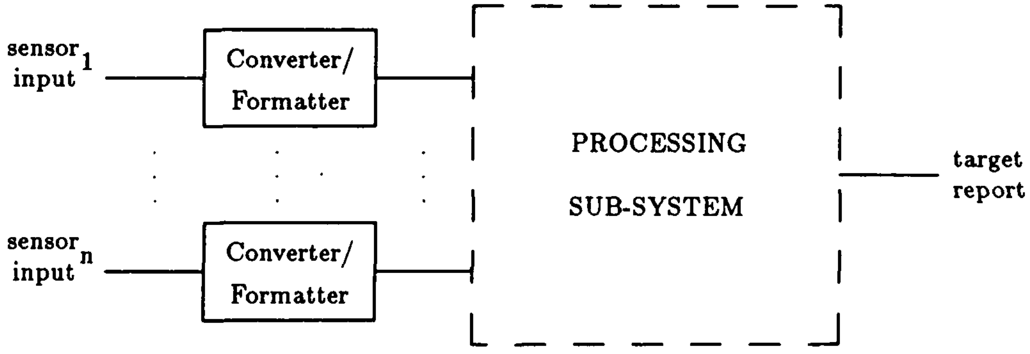


Figure 2.1: Application System Concept

The language developed in this work is used not only to denote the computational needs of certain signal processing functions, but also to describe the entire application system. The application system concept underlying the language constructs is depicted in Figure 2.1. The sensor input and converter/formatter are external to the signal processing functions considered here; this work is concerned with optimization of the processing subsystem. Specifically, optimization of the processing subsystem consists of mapping the signal processing functions to tasks in a generic multiprocessor architecture such that throughput, memory and latency design constraints are met.

The sensor input is a continuously varying value, *e.g.*, a voltage or energy level as provided by an antenna or receiver of some kind. The converter/formatter first performs receiver front-end processing and down conversion and then analog to digital conversion. The resulting data are then collected and formatted into a data item which will be referred to in this work as a *basic object. Basic objects are therefore composite pieces of data representing finite collections of samples from the continuous data source.*

An example of a basic object and its derivation from a real-life quantity can be found in the space-based radar application. Each received pulse is sampled N_r times. During each received pulse, the converter performs A/D conversion; at the end of each received pulse, the formatter formats the samples into a vector of size N_r. Since there are N_{rc} pulses in a burst, after each burst, the pulse vectors are formatted into the N_{rc} rows of an $N_r \times N_{rc}$ matrix. Each matrix is a basic object.

The output of the converter/formatter is a periodic stream of formatted values, in this case $N_r \times N_{rc}$ matrices. The frequency of the stream depends on how often data is collected and formatted. In the above example, if the pulse vectors were the basic objects on the output stream, then its frequency would be greater, since pulses are received more frequently than bursts (bursts are composed of a number of pulses). Each basic object in the stream is called a *token*; each token in a stream is an input to an invocation of a signal processing function inside the processing sub-system.

This stream of basic objects output from the converter/formatter is the *base input stream* as derived from the sensor input. The type of the basic objects and the frequency of the stream are used to derive the frequency and type for all streams inside the processing subsystem. If there is more than one base input stream and not all have the same frequency, then one particular input stream is designated *the* base input stream. The processing subsystem consists of signal processing functions which operate on basic objects. The basic object types defined to the processing subsystem are multi-dimensional arrays whose elements are numbers (in the degenerate case where the basic object is a scalar, the basic object is defined with dimension zero).

The signal processing functions are generally executed in a pipelined fashion, where the input to a stage in the pipeline comes directly from the output of a previous stage. Each token in the base input stream follows a processing "path" through the application pipeline; this path is called an *instance* of the pipeline. An instance of the pipeline consists of a sequence of function instances, or invocations, in which individual tokens in the stream are processed. Functions are therefore mapped onto input streams, yielding output streams which are composed of the tokens output from the function.

Different tokens from the input stream can be processed either in parallel disjoint instances of the entire pipeline or in common instances or some hybrid of the two where some of the functions in the pipeline are common and some are disjoint. A single token can also be processed in separate instances of the pipeline. The different configurations of function instances which are realizable within the pipeline are determined by the amount of parallelism present in the signal processing functions.

In the execution model for the processing system, each instance of a function in the pipeline is implemented by an actual task on an actual multiprocessor. A task can process one or more instances of a function in the pipeline. The task is the smallest unit of independent processing in the model, since the architectures targeted in this work are coarse-grain and loosely-coupled. The execution model is therefore based on a general, coarse-grain dataflow architecture, the details of the which are described in the next section [3].

2.2 Dataflow Model

The dataflow model is based on features of the AOSP architecture and operating system which are relevant to application task configuration. This section describes these features and their correspondence to the items which compose the application system model.

Tasks in the AOSP communicate via messages which contain data items. Each message consists of a data descriptor and a data value or set of values. The data descriptor identifies the data to both the operating system and the application task. Data descriptor information includes a pointer to the data (which is determined when the operating system allocates memory for incoming messages), the size and dimension of the data, the descriptor table entry (which identifies the data to the destination task), status and event number. Each message therefore contains a basic object which is the input to the function instantiated by the task.

The event number "tags" the data, *i.e.,* identifies the instance of an input variable within the application pipeline. Event numbers are assigned by the system input which interfaces directly with the sensors. A lower bound, upper bound and increment are specified for an application's system input. Event numbers are assigned starting with the lower bound and incrementing for each data value until the upper bound is reached, then beginning again with the lower bound. The event number is therefore a unique identifier of the instance of the application pipeline in which the given message is processed.

The operating system supports tagged-token dataflow between tasks in the following manner: the input messages to a task are correlated by event number and grouped into *events.* When all data values in an event have arrived at the destination task, the specified entry point corresponding to the variables in the event is added to the ready-to-run queue. The designer specifies the number of dataflow entry points for each task and the number of input values per entry point.

Events can be queued at the destination task; the designer can specify the queueing level for each entry point in the task. Upon completion of an entry point, the task must explicitly deallocate the memory used by the input messages. Sending of messages is done explicitly, via system service calls. Receiving of messages is performed by the operating system and precedes task invocation.

Since the model is used for signal processing systems with real time requirements, the rates at which data is transferred between tasks is a crucial factor in maintaining system robustness. A major problem which can occur in the above dataflow implementation is the synchronization of inputs to a task where the frequency of one input is much larger than that of another input. In this case, the asynchronous dataflow model has no mechanism to prevent queue overflow at the more frequent input and responsibility lies with the programmer or scheduler to insure robustness of code.

2.3 Application Examination

Given the above system and execution models, the structure of the signal processing functions determines how the processing subsystem is optimized. The processing subsystem is optimized by finding a task configuration which takes advantage of the parallelism present in the applications within the performance constraints imposed by the architecture. Generalization of the nature of the parallelism in the applications examined suggests four *types* of parallelism present at the coarse-grain level which will be used to determine possible instance configurations. This type of parallelism is referred to as *partitionability* and is classified into the following categories:

- Independent Channel Processing
- Split-Stream Processing
- Temporal Partitioning
- Spatial Partitioning.

These are presented in detail in the following sections.

2.3.1 Independent Channel Processing

In many applications, base input streams from several input sensors are processed independently of each other, *i.e.,* in separate instances, for some or all of the application pipeline. An example of this occurs in the radar application, in which data from each channel is sent through a sequence of functions before results are merged after pulse-doppler processing and before magnitude and video integration. This type of parallelism is exploited by instantiating the individual processing sequences as separate sets of tasks, sending data from a given sensor to the corresponding task. If the number of processing elements in the network limits the number of tasks, then one task can process all sensor data as long as data is labelled with its originating sensor and the task keeps any necessary state of computation for any sensor separate from that of the other sensors.

The optimal configuration of tasks in this case may be a hybrid of these two extremes, where we partition the set of input channels into subsets and process each subset in a common task. The designer must determine the level of partitioning, *i.e.*, how many subsets constitute an optimal implementation. If the cost of communication is prohibitively high, a lower level of partitioning will yield better performance, *i.e.*, fewer subsets.

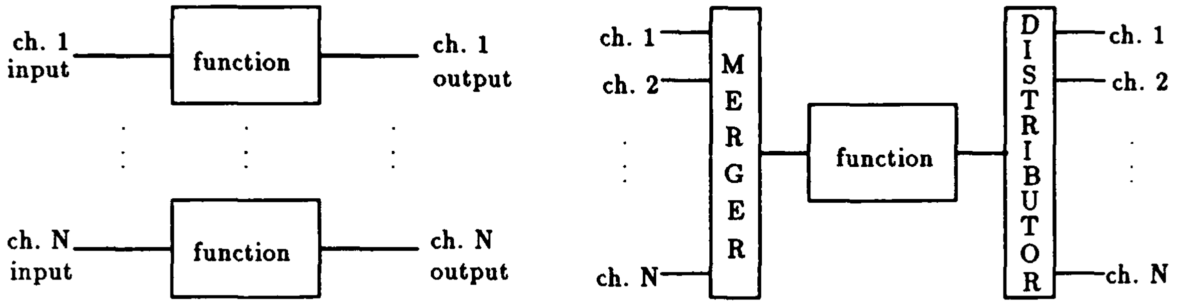


Figure 2.2: Independent Channel Processing Task Configurations

Sample task configurations are depicted in Figure 2.2. In the partition on the left-hand side, data streams from the different channels are processed in separate tasks. Each task instantiates the same function, therefore, another possible configuration consists of one task which processes the data from all input channels. This second configuration is depicted in the right-hand side of the figure. The MERGER in this figure consists of a mechanism combining into one output stream the tokens from the different input streams. The MERGER may be a non-deterministic merge or a simple rotation of input values in which a token is taken from each input stream in an ordered sequence, or cycle.

The DISTRIBUTOR in the figure consists of some mechanism by which the output of the function are returned to their appropriate processing sequence so that subsequent processing can be performed. The DISTRIBUTOR must therefore be able to distinguish between the output values so that the tokens will be sent to the correct output stream. If the MERGER and DISTRIBUTOR can be guaranteed to perform correctly, then *the two partitions are equivalent functionally* and differ only in task structure.

2.3.2 Split-Stream Processing

In many cases, two completely independent functions are performed on the same sensor output stream and therefore can be performed in parallel. An example of this occurs in some radar applications where there is jammer nulling. In this case, data streams from the main channels would be split into two streams. Data from one stream would be used to calculate the effect of the jammer. The jammer would then be subtracted out of the original signal, at which point the two streams are combined.

In many cases, the split streams are processed at different rates. Thus, the stream of result tokens has a different data rate from that of the input stream. In these cases, there must be some mechanism of insuring that relative frequencies of the streams to be combined are the same, in order to prevent queue overflow at a later input.

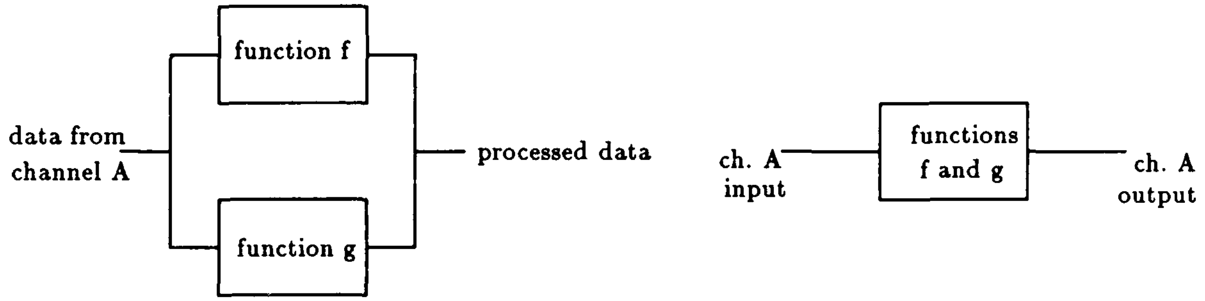


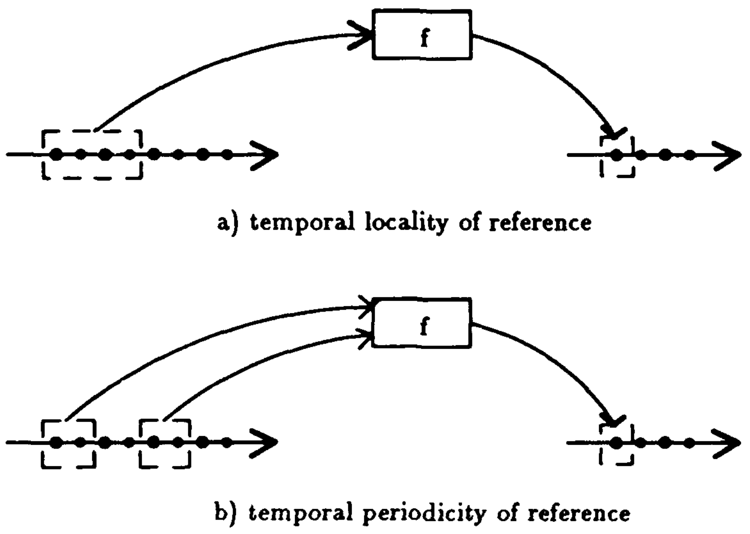
Figure 2.3: Split-stream Processing Task Configurations

We can take advantage of the parallelism here by instantiating each separate processing sequence as a set of tasks arranged in a pipeline. This is the simplest type of parallelism to identify and exploit; circumstances which would prohibit a maximally parallel implementation include a high communication latency and/or a small number of nodes in the network. Task configuration is depicted in Figure 2.3.

2.3.3 Temporal Partitioning

Many signal processing functions exhibit history-sensitive behavior, *i.e.*, outputs are functions of current and past inputs. If the function is *not* history-sensitive, then a data value from the stream can be processed independently of those preceding it. Thus, all tokens from the input stream can be processed in parallel. We can take advantage of this parallelism by instantiating several functions and distributing values from the stream to the different tasks as they arrive. The number of parallel tasks in the optimal configuration depends on the computational latency of the function.

If the function is history-sensitive, then it has some *state* which must be maintained between successive tokens from the input stream. This state is initialized in the first invocation of the function and modified upon each subsequent invocation. For some functions, the state is periodically re-initialized and therefore the function output depends on a finite portion of the input stream, *i.e.*, a finite set of successive data tokens called a *temporal window*. The values in a temporal window are processed as a group to yield a result token for the output stream. When functions exhibit this type of history-sensitive behavior, they possess the property of *temporal locality of reference*. This property is depicted in Figure 2.4a. In this figure, if the input stream



a) temporal locality of reference

b) temporal periodicity of reference

Figure 2.4: Temporal Locality and Periodicity of Reference

is $[a_0, a_1, a_2, \ldots]$, then the output stream is $[b_0, b_1, b_2, \ldots]$ where $b_i = f(a_0, a_1, a_2, a_3)$ and the window is composed of four successive values from the stream. The function f is applied an infinite number of times, since streams are infinite and the state size is finite.

For other functions, the state is maintained between sets of successive tokens which occur at regular intervals in the stream, *i.e.*, sets of values that are periodically *sampled* from the stream. The values in each set, or window, are processed as a group to yield a result token for the output stream. This result token is "remembered" by the function for processing of the next window. The function output therefore depends on an infinite sequence of finite windows. When functions exhibit this type of history-sensitive behavior, they possess property of *temporal periodicity of reference*. This property is depicted in Figure 2.4b. In this figure, f is applied to all windows of size two with state maintained between windows every four tokens in the stream. The state, or history, of the function f is maintained between a finite number of invocations of f, but is infinite in size. The function f is called a *base-level function* in this context.

An example of this occurs in the space-based radar application in which video integration is performed on six successive matrices from the stream, then repeated for each group of six matrices. The six matrices are summed and the result is "put" on the output stream. Thus the output of this function has a smaller frequency than the input, since one matrix is output for every six which are input. Another example is the pulse doppler function, in which five matrices are collected from the input stream and combined into one matrix. The FFT is then performed on every column of the resulting, combined matrix. The output frequency of this function is also smaller than the input frequency, since one result matrix is output for each five input.

Some functions may also operate only on selected (or sampled) groups of values from the stream and are not invoked on the other values. In these cases, the functions exhibit temporal locality and periodicity of reference, however, processing is not repetitive. In other cases, there is overlap between the groups of values extracted from the input stream. An example of this would be a function in which one token is output for each input, but the state is composed of a finite number of previous input tokens, so there is a finite-size state involved even though the frequency does not change.

The potential parallelism arises in cases where the processing window is of finite size. In these cases, the parallelism can be exploited by routing *groups* of tokens from the input stream to different tasks in a cycle much like individual tokens are routed in the non-history-sensitive case. The number of tokens in each group is the size of the state of the function. If there is overlap between groups of values, then there will be an extra communication cost. If the state depends on tokens selected at regular intervals from the stream, then there is an upper bound on the number of parallel tasks in the partition, since some groups of values must be routed to the same task.

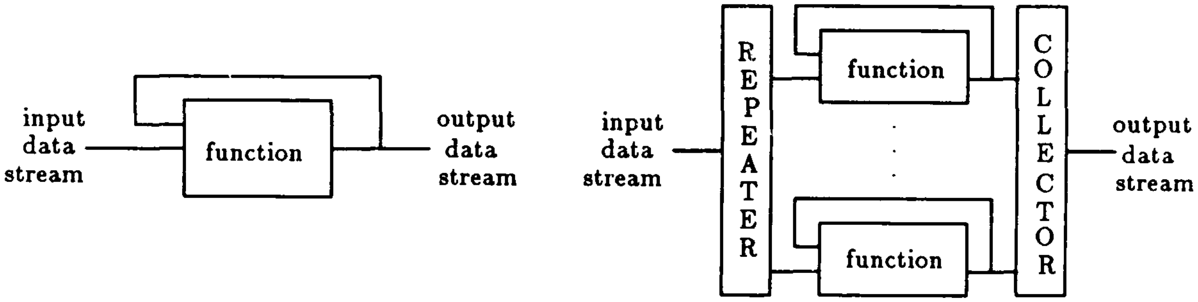


Figure 2.5: Temporal Partitioning Task Configurations

Sample task configurations are depicted in Figure 2.5. In the partition on the left side, the function is contained in one task which maintains its own state. In the partition on the right side, some number of parallel tasks perform the same function as the original configuration. The REPEATER consists of some mechanism by which groups of values are selected from the stream and routed to the tasks in a cycle. The COLLECTOR consists of some mechanism by which the outputs of the different tasks are combined into one stream.

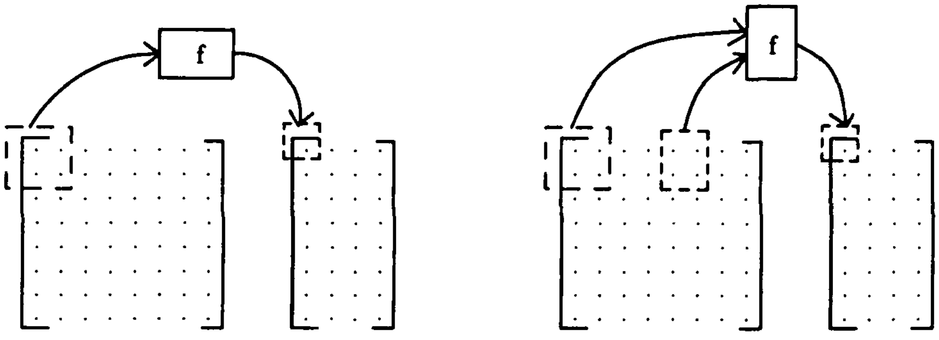
In order to calculate the throughput of each task in the configuration, we need to know the latency of the base-level function and the frequency with which that function will be applied to process a given temporal window. The frequency with which the function will be applied depends on how frequently the temporal windows are selected from the stream. The frequency of temporal windows depends on window size, inter-window spacing, window overlap and the number of tasks in the configuration. If we assume that the base-level function always outputs one token per window, then the frequency of application is the frequency of the output stream.

In order to maintain consistency, a predefined order must be maintained between REPEATER and COLLECTOR. If there is overlap of outputs, then there must be some means of merging overlapped values, *i.e.*, an averaging, or summing. In this work, functions are defined to output only one token for each input group, therefore functions exhibit no overlap of output tokens. This aspect of the COLLECTOR is therefore not considered. If the REPEATER and COLLECTOR can be guaranteed to maintain the *ordering* of the input and output streams, then *the two partitions are equivalent functionally* and differ only in task structure.

2.3.4 Spatial Partitioning

Many signal processing functions exhibit regular patterns of reference *within* a large piece of multidimensional data which can be exploited to increase parallelism in implementing the function. Functions in this context are viewed as transformations from one basic object to another in which some section of the output data is a function of a corresponding section of the input data. The simplest example is a function which multiplies each element of a matrix by a constant and outputs the resulting matrix. In this case, each element of the output matrix is a function of the corresponding element of the input matrix. Since processing of each element is independent of processing of the other elements in the input matrix, a maximally parallel implementation would consist of a separate task for each element.

In other functions, the patterns of access are regular enough to exploit parallelism, but not as simple as in the previous example. For example, a function may operate on each column of an input matrix, produce an output vector for each column and then combine these vectors as rows or columns of an output matrix. In another case, a given element of the output matrix is derived by applying a lower-level function to a *window* of elements around the corresponding element of the input matrix. When functions exhibit these patterns of behavior, they possess the property of *spatial locality of reference*. This property is depicted in Figure 2.6a. In this figure, if $A = a_{ij}$



a) Spatial locality of reference

b) Spatial periodicity of reference

Figure 2.6: Spatial Locality and Periodicity of Reference

is the input matrix and $B = b_{ij}$ is the output matrix, then $b_{ij} = f(a_{ij}, a_{i+1,j}, a_{i,j+1}, a_{i+1,j+1})$ and the window is a 4×4 region of the input matrix.

For other functions, sections of the output matrix are derived by applying lower level functions to windows of elements of the input matrix at regular intervals on a given dimension. This is similar to temporal periodicity of reference, however, sampling is performed on a given dimension instead of in time. When functions exhibit these patterns of behavior, they possess the property of *spatial periodicity of reference.* This property is depicted in Figure 2.6b. In this figure, if $B = b_{ij}$ is the output matrix, then b_{ij} is f applied to all 4×4 windows separated by four elements on the second (column) dimension. The function f in this context is called a *base-level function.*

Some functions may also operate only on selected values from the input matrix and are not invoked on the other values. In these cases, the selected windows of values exhibit spatial locality and periodicity of reference, however, processing is not repetitive. In other cases, there is overlap between the windows of the input matrix and/or the windows of the output matrix. If it is the latter case, then combining the output data requires more processing.

Examples of spatial locality of reference occur in many of the space-based radar application functions. One example is the pulse compression procedure, in which an FFT is performed on each row of the input matrix independently. The resulting vectors are the rows of the output matrix. In another kind of application, a tracking radar, spatial locality is exhibited in a tracker which divides a sensor field of view into separate windows and processes each independently. The target reports from these windows are combined into one overall target report. In each case, several tasks can execute the same function on the sections of data in parallel, assuming a small processing overlap.

In order to exploit this type of parallelism, we can instantiate several versions of the function as separate tasks and send a section of the original input data to each task. The optimal configuration of tasks, *i.e.*, the number of sections which yields the highest throughput, depends on the computational latency of the base-level function and the communication latency between tasks. The size of the input to each task will depend on the number of parallel tasks. Each task will perform the base-level function on all the window(s) in the input section. A maximally parallel task configuration would process each window of the input data in a separate task.

Communication latency will be a function of inter-section dependency. If the sections are completely disjoint, no communication between tasks is necessary. The number of tasks in the optimal configuration will therefore not be limited by these communication costs. If sections overlap, communication costs will be higher and the optimal configuration may consist of fewer tasks with higher computational latencies.

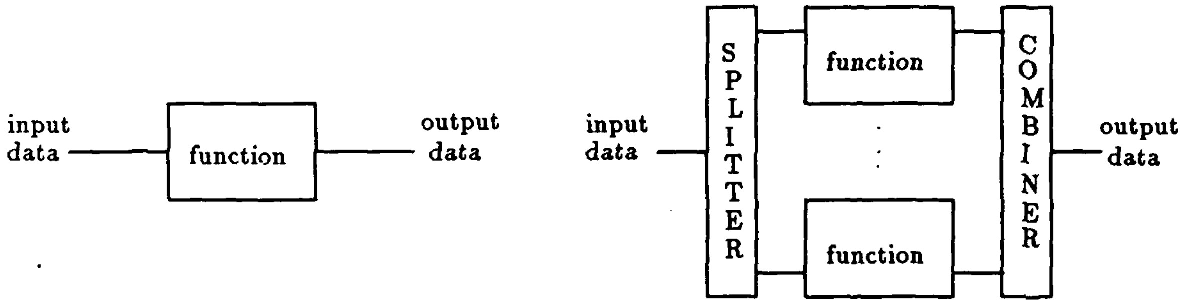


Figure 2.7: Spatial Partitioning Task Configurations

Sample task configurations are shown in Figure 2.7. In the partition on the left-hand side, the entire input matrix is processed in one task, which repeats some base-level function for each window of the input matrix. In the partition on the right-hand side, some number of parallel tasks perform the same function as the original configuration on a section of the original input data. The SPLITTER consists of some mechanism by which the input matrix is divided into some number of sections. These sections may be composed of one or more processing windows, depending on the size of the section. The COMBINER consists of some mechanism by which the outputs of the different tasks are combined into one large output matrix which is the same as the output matrix in the first partition.

In order to calculate the latency and throughput of each task in the configuration, we need to know the latency of the base-level function and the number of times that function will be applied to process a given section. The number of times that the function will be applied in the processing of a given section is the number of processing windows contained inside the given section. The number of processing windows inside a section is a function of window size, section size, window overlap and inter-window spacing (if present). The size of the input to the base-level function is the window size if there is no inter-window spacing; if there is inter-window spacing, then the size of the base-level function input is the total size of all regularly spaced windows inside the input data matrix.

An example is the following situation in a space-based radar application: a filter function is applied to each column of a matrix. If the matrix is split into two sub-matrices along the second (column) dimension and each half is processed in a separate, parallel task, then the filter function is applied half as many times in each task as in a task which implemented the filter function on each column of the original matrix. Given a section consisting of half of the matrix, we can derive the number of columns (windows) in the section. If we know the latency of the filter for a given column, then we know the latency for each task.

In order to maintain consistency, a predefined correlation must be maintained between SPLITTER and COMBINER. If there is overlap of outputs, then there must be some means of merging overlapped values. In this work, this merging is assumed to be an averaging and processing costs are calculated with this assumption. If the SPLITTER and COMBINER can be guaranteed to maintain the correct *ordering* of the input and output sections, then *the two partitions are equivalent functionally* and differ only in task structure.

2.4 Program Graph Representation

Given the above system, execution and application models, the role of a programming language and intermediate representation must be defined in accordance with the goals outlined in the introduction to this thesis. A language is necessary for the programmer to code the application; however, details of the task structure, *i.e.,* the execution of the application, should not concern the programmer. Similarly, at decomposition time, we would like to have a representation in which we can manipulate only task structure in order to find the best partition without worrying about functionality. We would like the language to express the parallelism in the application, as well as compile efficiently into the intermediate representation.

At decomposition time, we therefore would like a single intermediate representation to represent a variety of task configuration options. If this is possible, then we can use the same representation to "try" out an option, evaluate it, then try another if that isn't "good" enough without having to change the functionality, *i.e.*, without having to re-code or re-compile the application. The different task configuration options considered in this work are the ones outlined in the previous section. These determine the nature of the intermediate representation, which in turn determines the nature of the language.

The intermediate representation developed here is based on a dataflow representation, since both the execution and application models are based on dataflow techniques. A dataflow representation of an application consists of a directed graph in which nodes are functions and arcs define data dependencies between functions. A function executes when its inputs "arrive" at the node, where each input has a queue of specifiable size to hold values waiting to be matched. The pipelined nature of signal processing can easily be captured in this representation, which also corresponds directly to the dataflow model defined above.

The approach taken in this work to solving the representation problem outlined above is to use a dataflow representation of an application, included in which are well-defined constructs with pre-defined, special significance in the graph. These constructs will denote the behavior of the supplementary mechanisms defined in the previous section, specifically the MERGER and DISTRIBUTOR, REPEATER and COLLECTOR and SPLITTER and COMBINER. These constructs, together with the corresponding function, form a partition of the original function. If these constructs accurately describe the correct "routing" mechanisms described previously, then the partition is *functionally equivalent* to the original function, and can replace it in the graph.

Since a routing mechanism with the necessary behavior can be constructed for any desired number of tasks in the partition, the graph constructs denoting the mechanisms can be *parameterized.* The values supplied for the parameters determine the number of parallel tasks used to implement the original function, and therefore determine a task configuration. If all task configurations realizable on the execution model can be described by a partition, then the partition is *complete.* By designing the special graph constructs correctly, all partitions denoted in the graph will be both functionally equivalent and complete. Conversely, for a given hardware architecture and the kinds of parallelism described, any functionally equivalent and complete partition can be denoted in the graph.

In order for the special graph constructs to have meaning in the context of a dataflow representation of the application, the semantics of nodes and arcs in the graph must be redefined. Arcs in the program graph denote a set of parallel streams which can vary in size. Nodes in the program graph represent abstract functions which are actually implemented by the number of parallel tasks in the partition of the function. The number of parallel tasks is given by the number of parallel streams in the input arc to the function. Each task represented by the node is invoked on a different stream in the input arc. The number of parallel streams in the input arc is determined by the parameter values supplied to the special, parameterized construct immediately preceding the function. Since these constructs can be nested, they are also implemented by a number of parallel tasks determined by the size of the input arc to the construct.

These special constructs used to denote the routing mechanisms are called the *Data-Routing Operators*, and are described in detail in the Chapter 4. Since the data-routing operators change the data on a stream and the frequency of the stream, we must have two fundamental guarantees with respect to any program graph:

- consistency is maintained, *i.e.*, wherever we manipulate an input stream to describe a new task configuration for a function, we must manipulate the output streams to reflect the change in task structure without changing functionality.
- asynchronicity of dataflow computation is preserved without breakdown of the execution model, *i.e.*, without queue overflow.

The approach to assuring consistency utilized in this work is to *encapsulate* base-level functions in the graph with data-routing operators in *pairs*. The first operator in the pair splits an input stream into some number of streams which determines the number of tasks in the partition. The second operator in the pair combines the output streams of the tasks in the partition into one stream.

The encapsulated functions are may be substituted for the original functions and therefore assure that consistency is maintained. If consistency is maintained in this manner, functional equivalence of partitions is assured. Maintaining consistency in the graph by inserting pairs of operators achieves functional equivalence of partitions because of the general property of *substitutability*. This property is defined below along with the necessary definitions.

Definition 2.1 (Routing Functions) *A pair of functions* (p, q) *is a* routing function pair *iff*

$p(M_1) = \{s_1, \ldots, s_k\}$ and $q(\{t_1, \ldots, t_k\}) = M_2$ for any k, where each s_i and each t_i is a stream of basic objects and M_1 and M_2 are also streams of basic objects.

Definition 2.2 (Partition Structure) *A* partition structure *is a function application of the form $q(S(p(M)))$ where (p,q) is a routing function pair and $S(p(M)) = \{f_1(s_1), \ldots, f_k(s_k)\}$ where $f_i(s_i)$ is the stream of objects resulting from the application of function f_i to the object stream $s_i \in p(M)$.*

Definition 2.3 (Partition) *A* partition structure *$q(S(p(M)))$ is a* partition *of a function F if $F(M) = q(S(p(M)))$ where $F(M)$ is the stream of objects resulting from the application of F to the object stream M.*

Property 2.1 (Substitutability) *A partition of a function F is* substitutable *for F.*

In the context of the program graph, the data-routing operators act as the partitioning functions and their inverses. Therefore, consistently inserting pairs of operators into the graph achieves functional equivalence of partitions. Substitutability is maintained independently of the level of partitioning. This means that if a partition of a function is substitutable for the function, then it should be substitutable for any k, where k is the number of independent concurrent tasks which will replace the original function.

Given the above graph representation, we will then need a language which compiles efficiently into a consistent program graph. The language $\mathcal{PGL}$ contains special constructs which compile directly into data-routing operators and function nodes. Compilation of $\mathcal{PGL}$ code into a graph is achieved by a set of Transformation Rules for transforming $\mathcal{PGL}$ programs into graphs. The transformation rules are designed such that any graph produced from a $\mathcal{PGL}$ program by the transformation rules is consistent.

Given the above specifications, when coding a function in $\mathcal{PGL}$, the programmer need not specify task structure but instead concentrates only on functionality of his code. Similarly, when designating the task structure from the graph, *i.e.*, when decomposing an application, the optimization algorithm manipulates only the operator parameters which do not alter functionality. The $\mathcal{PGL}$ code and the program graph therefore have well-defined roles in application development; the former is responsible solely for expressing the functionality of an application, and the latter is responsible only for expressing the task structure correctly. The correspondence between the two representations insures that manipulation in one domain maintains correctness in the other.

Automatic partitioning of the application is enabled by restricting all parallelism exploited in the mapping of functions to tasks to that which is conveyed by the placement of operators in the graph. Since all graph constructs denote well-defined behavior, each combination of operator parameters in the graph denotes a different partition of the application which can be "evaluated". Optimization algorithms, whether iterative or partially interactive, can be developed as further research. This work is concerned with finding a suitable representation; the actual algorithms are subject for further research.

In the model, communication latency and token processing overhead are large. Thus, the major design goal in partitioning an application is to minimize the communication-to-computation ratio while maximizing throughput. The graph and language constructs are therefore designed for coarse-grain dataflow, in which the amount of processing within a task is large enough to compensate for communication delays. This means that base-level functions or code blocks are the smallest unit of scheduling. In fine-grain dataflow, the machine-level instruction is the smallest unit of scheduling.

The complexity involved in programming with the dataflow paradigm lies in mapping the dataflow graph to the target architecture represented by the dataflow model. In the static dataflow model, partitioning of the application into tasks occurs before the mapping of code to processors. Applications are partitioned according to one or more design criteria, *e.g.,* maximum speedup, minimum communication cost, minimum latency, maximum throughput, *etc.*

Approaches to the mapping problem involving different tools and algorithms to automate the mapping process can be found in the literature [7, 10, 14, 16, 21, 27]. In many cases, graph-like languages are used to expedite the process [18]. Partitioning of the application into the tasks which are mapped to processors is generally done by hand. The program graph and language described in this work represent an approach to automating the partitioning process. The ultimate goal of this approach is to include information about hardware configuration in the partitioning process, so that the subsequent mapping problem can be solved more efficiently.

2.5 General Language Requirements

In order to support the automatic partitioning discussed above, we need a language which will express the parallelism inherent in the application at the *functional* level. Functions are defined as filters that operate on continuous streams of large pieces of data. The compiler should be able to detect any parallelism in the code without needing explicit denotation.

The language must contain special constructs to achieve this goal. The language $\mathcal{PGL}$, defined in the next chapter, is based on the language Id [22] and has the same syntax, but is extended to include the extra constructs required to establish a one-to-one mapping to graph constructs. $\mathcal{PGL}$ is based on a filtered stream concept similar to that of LUCID [26], and proposed extensions to VAL [20].

In addition to the needs outlined above, the language must contains features appropriate for a modern programming language, namely procedure abstraction, data abstraction and a type system. These are described in the next sections.

2.5.1 Procedure Abstraction

Procedure abstraction is a fundamental requirement of most modern languages [19]. Procedure abstraction enables the modular development of programs, so that each procedure is compiled individually. If one procedure is defined to be substitutable for another, then all procedures which invoke the procedure should compile correctly when the procedure is recompiled.

2.5.2 Data Abstraction

Data abstraction is also a fundamental requirement of a modern language. The representation of data should be hidden by the language, which should only provide operations for a given data structure.

Inputs and outputs to procedures and operators in $\mathcal{PGL}$ are data structures referred to as objects. Objects are scalars or matrices. Matrices can be split into smaller pieces, called *windows* for repetitive processing or iteration. Matrices therefore have operations for selecting, replacing, inserting and iterating over windows of elements.

$\mathcal{PGL}$ provides a set of these operations which are independent of matrix size but dependent on matrix rank. $\mathcal{PGL}$ also provides a lower-level mechanism for defining new operations on matrices which necessitate defining windows for selection. The programmer can therefore concern himself with details of representation if he desires patterns of access to data structures which are not provided in the $\mathcal{PGL}$ library of operators.

Abstract datatypes are not yet supported in $\mathcal{PGL}$. Abstract datatypes will have iteration and selection operations defined on them in terms of the abstract datatype and not the underlying representation. Implementation of abstract datatypes is left for future work.

2.5.3 Type System

A strong type system is also a fundamental requirement of a modern language. A strong type system prevents errors at run time and enables optimization of compiled code.

In $\mathcal{PGL}$, type inference is used to determine the type of intermediate datatypes. Since operators on objects are rank-dependent, in $\mathcal{PGL}$, the type of a matrix refers to its rank. Therefore, type information for matrices is size-independent. The element-type of a matrix is inferred from the context.

Since procedures are also intermediate functions in an application pipeline, their inputs and outputs can also be inferred from a base input stream definition. Since matrix operators are rank-dependent, the type of any object can be inferred from the context in which it appears.

Chapter 3

$\mathcal{PGL}$

Programs in $\mathcal{PGL}$ are collections of definitions which describe an application pipeline. The applications for which $\mathcal{PGL}$ is designed reflect the high-level system model developed in Chapter 2. Presentation of $\mathcal{PGL}$ will begin with a description of the objects defined in $\mathcal{PGL}$ and the operations which can be performed on them. From this, a well-defined type system emerges, with type checking performed by *type inference.*

Since all functions in the system model operate on periodic streams of data objects, we will need a language concept to describe streams. After defining basic objects, we define streams in $\mathcal{PGL}$ and the operations which can be performed on them. Once these operations are defined, we can then define functions which operate on streams and procedures, which are the basic building blocks for application code. Finally, we describe iterations. which are the main expressions of parallelism in $\mathcal{PGL}$, leading to a complete language for describing the applications based on the model in Chapter 2.

The $\mathcal{PGL}$ language contains special constructs designed to express the kinds of parallelism outlined in the previous chapter. These constructs will translate directly into program graph constructs. The syntax of $\mathcal{PGL}$ closely resembles that of Id [22]. $\mathcal{PGL}$ is presented in the first sections of this chapter. The $\mathcal{PGL}$ code for the entire space-based radar application is presented in Appendix C.

3.1 Basic Objects

The basic objects defined in $\mathcal{PGL}$ are scalars (numbers), and matrices. Examples of scalars are integers, complex and floating point numbers. Matrices are multidimensional, composite objects whose elements are scalars. Matrices have rank r and size $(n_1, n_2, \ldots, n_r)$ where n_i is the size of the matrix on dimension i. Scalars are viewed as matrices with rank equal to zero. The type notation for objects is presented below.

Object Class	*Type Notation*
Basic Objects	o
Integers and Real Numbers	n
Complex Numbers	c
Matrices	m

3.2 Operations on Objects

$\mathcal{PGL}$ includes a set of operators which perform operations on basic objects. These fall into three categories: operations on scalars, operations on matrices and special operations on matrices called *spatial selection operations.*

3.2.1 Operations on Scalars

$\mathcal{PGL}$ contains operators for arithmetic operations on each type of number. For integers and real numbers, the operators are denoted by their symbols $(+, -, *, /)$. These operators are written in binary infix notation, *e.g.,* $3 + 4, a + b$, and have signature $n \times n \rightarrow n$. For complex numbers, these operators are the functions **`complex-add`**, **`complex-sub`**, **`complex-mult`** and **`complex-div`** with signature $c \times c \rightarrow c$. Function application is written

function-name input-0 ... input-n

where *function-name* is a function of arity n.

Operator precedence is according to standard arithmetic practice. There are also operators to convert integers and real numbers to complex numbers; these are the functions **`real-to-complex`** and **`integer-to-complex`** with signature $n \rightarrow c$. There are also operators which implement abstractions for dealing with complex numbers; these are the functions **`real-part`**, **`imag-part`**, **`complex-mag-sq`** and **`complex-mag`** with signature $c \rightarrow n$ for real part, imaginary part, complex magnitude squared and complex magnitude, respectively. Each operator compiles into a node in the program graph with predefined values for the parameters.

3.2.2 Operations on Matrices

$\mathcal{PGL}$ contains operators for defining primitive operations on matrices of rank one and rank two, *i.e.*, vectors and two-dimensional arrays. Some of these are borrowed from the language Id [22]. These are the **`fold_nD_array`**, **`map_fold_nD_array`** and array comprehension constructs. Each of these operators compiles into a node in the program graph with parameter values derived from analysis of the Id construct. Three other matrix operations are included as basic operations. These are the **`fft`** and **`fft_1`** functions on vectors and the **`inverse-matrix`** function on two-dimensional arrays. These implement the FFT, inverse FFT and inverse matrix functions, respectively. Future versions of $\mathcal{PGL}$ and the compiler will add new constructs for dealing with these special cases. Operations on matrices have signature $m \times \ldots \times m \rightarrow m$.

3.2.3 Spatial Selection Operations

$\mathcal{PGL}$ contains operators for a special class of operations on matrices called *selection* operations. Some of these operators, the selector operators, define a region of the input matrix and output an object formed by extracting the region from the matrix. Other operators, the replacer operators, define a region of an input matrix and output the same matrix with the defined region replaced by another basic object of the appropriate size and shape. Other operators, the inserter operators, define a place in an input matrix and output the same matrix with another basic object inserted into the defined place.

Selectors

Selection of regions within a matrix occurs in regular patterns, therefore we can generalize selection in one construct. Some of the ways in which we will want to select portions of a matrix are indicated below. If the input matrix is

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix}$$

then the following list shows some of the regions that may be selected and the basic objects that result from selection.

$$a_{35}, \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{41} \\ a_{51} \\ a_{61} \\ a_{71} \\ a_{81} \end{bmatrix}, \begin{bmatrix} a_{23} & a_{24} \\ a_{33} & a_{34} \end{bmatrix}, \begin{bmatrix} a_{21} & a_{25} \\ a_{41} & a_{45} \\ a_{61} & a_{65} \\ a_{81} & a_{85} \end{bmatrix}, \begin{bmatrix} a_{13} & a_{14} & a_{17} & a_{18} \\ a_{23} & a_{24} & a_{27} & a_{28} \\ a_{53} & a_{54} & a_{57} & a_{58} \\ a_{63} & a_{64} & a_{67} & a_{68} \end{bmatrix}$$

Each possibility above is derived by applying a *spatial mask* to the input matrix. In order to generalize all above possibilities, we must identify the components of each mask.

The mask in each case consists of a basic *window*, or shape, an offset (number of matrix elements before the first in the mask) on each dimension and an inter-window spacing for cases where there is sampling on a dimension. In the first case, the window is one element (1×1), the offsets are 2 and 4 on dimensions 1 and 2, respectively, and the spacing is zero, since there is no regular sampling. The description of a region will therefore consist of a list $((o_1, l_1, d_1), \ldots, (o_r, l_r, d_r))$ where r is the rank of the matrix. For a given dimension i, o_i denotes the initial offset of the selected region, l_i denotes the size of the shape and d_i denotes the inter-window spacing. If any l_i is zero, then the size of the window is the size of the matrix on dimension i. If any d_i is zero, then there is no inter-window spacing on dimension i.

For the list of regions displayed above, the mask definitions are as follows: $((2,1,0),(4,1,0))$, $((0,8,0),(0,1,0))$, $((1,2,0),(2,2,0))$, $((1,1,1),(0,1,3))$, and $((0,2,2),(2,2,2))$, respectively.

A spatial selector operator on matrices must therefore have a defined window to select from the input matrix. Selector operators are defined in $\mathcal{PGL}$ using the `def-select` construct:

`def-select` *selector-name* = `make-select` *window;*

where *window* is the list defined above. This statement binds *selector-name* to a function with signature $m \rightarrow m$. $\mathcal{PGL}$ contains several built-in spatial selectors; these are listed in a later section.

Replacers

Replacement of regions within a matrix occurs in regular patterns, therefore we can generalize replacement in one construct. Some of the ways in which we will want to replace portions of a matrix are depicted below. If the input matrix and replacement object are

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix}, \quad \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

repectively, then the following list shows some of the basic objects which might result from replacement.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ a_{21} & b_{11} & b_{12} & b_{13} & b_{14} & a_{26} & a_{27} & a_{28} \\ a_{31} & b_{21} & b_{22} & b_{23} & b_{24} & a_{36} & a_{37} & a_{38} \\ a_{41} & b_{31} & b_{32} & b_{33} & b_{34} & a_{46} & a_{47} & a_{48} \\ a_{51} & b_{41} & b_{42} & b_{43} & b_{44} & a_{56} & a_{57} & a_{58} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix}, \quad \begin{bmatrix} a_{11} & b_{11} & a_{13} & b_{12} & a_{15} & b_{13} & a_{17} & b_{14} \\ a_{21} & b_{21} & a_{23} & b_{22} & a_{25} & b_{23} & a_{27} & b_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{51} & b_{31} & a_{53} & b_{32} & a_{55} & b_{33} & a_{57} & b_{34} \\ a_{61} & b_{41} & a_{63} & b_{42} & a_{65} & b_{43} & a_{67} & b_{44} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix},$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{31} & a_{32} & b_{33} & b_{34} & a_{35} & a_{36} & b_{37} & b_{38} \\ a_{41} & a_{42} & b_{43} & b_{44} & a_{45} & a_{46} & b_{47} & b_{48} \\ a_{51} & a_{52} & b_{53} & b_{54} & a_{55} & a_{56} & b_{57} & b_{58} \\ a_{61} & a_{62} & b_{63} & b_{64} & a_{65} & a_{66} & b_{67} & b_{68} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix}$$

Each possibility above is derived by fitting the replacement object into a region of the input matrix defined by a mask. Masks for replacement are defined in exactly the same manner as masks for selection, however, instead of a region to be selected, the mask identifies a region to be replaced. For the list of replaced regions displayed above, the mask definitions are as follows: $((1,4,0),(1,4,0)),((0,2,2),(0,1,1)),((2,4,0),(2,2,2))$, respectively.

A spatial replacer operator on matrices must therefore have a defined window to replace on the input matrix. Replacer operators are defined in $\mathcal{PGL}$ using the **def-replace** construct:

def-replace *replacer-name* = **make-replace** *window*;

where *window* is the list defined above. This statement binds *replacer-name* to a function with signature $m \times m \rightarrow m$. $\mathcal{PGL}$ contains several built-in spatial replacers; these are listed in a later section.

Inserters

Insertion of objects into a matrix occurs in regular patterns and therefore we can generalize insertion into one construct. Some of the ways in which we will want to insert objects into a matrix are depicted below. If the input matrix and insertion object are

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix}, \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \\ b_{51} & b_{52} & b_{53} & b_{54} \\ b_{61} & b_{62} & b_{63} & b_{64} \\ b_{71} & b_{72} & b_{73} & b_{74} \\ b_{81} & b_{82} & b_{83} & b_{84} \end{bmatrix}$$

respectively, then the following list shows some of the basic objects which might result from insertion.

$$\left[\begin{array}{cccccccccccc} a_{11} & a_{12} & a_{13} & a_{14} & b_{11} & b_{12} & a_{15} & a_{16} & a_{17} & a_{18} & b_{13} & b_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} & b_{21} & b_{22} & a_{25} & a_{26} & a_{27} & a_{28} & b_{23} & b_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} & b_{31} & b_{32} & a_{35} & a_{36} & a_{37} & a_{38} & b_{33} & b_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} & b_{41} & b_{42} & a_{45} & a_{46} & a_{47} & a_{48} & b_{43} & b_{44} \\ a_{51} & a_{52} & a_{53} & a_{54} & b_{51} & b_{52} & a_{55} & a_{56} & a_{57} & a_{58} & b_{53} & b_{54} \\ a_{61} & a_{62} & a_{63} & a_{64} & b_{61} & b_{62} & a_{65} & a_{66} & a_{67} & a_{68} & b_{63} & b_{64} \\ a_{71} & a_{72} & a_{73} & a_{74} & b_{71} & b_{72} & a_{75} & a_{76} & a_{77} & a_{78} & b_{73} & b_{74} \\ a_{81} & a_{82} & a_{83} & a_{84} & b_{81} & b_{82} & a_{85} & a_{86} & a_{87} & a_{88} & b_{83} & b_{84} \end{array}\right],$$

$$\left[\begin{array}{cccccccccccc} a_{11} & a_{12} & b_{11} & a_{13} & a_{14} & b_{12} & a_{15} & a_{16} & b_{13} & a_{17} & a_{18} & b_{14} \\ a_{21} & a_{22} & b_{21} & a_{23} & a_{24} & b_{22} & a_{25} & a_{26} & b_{23} & a_{27} & a_{28} & b_{24} \\ a_{31} & a_{32} & b_{31} & a_{33} & a_{34} & b_{32} & a_{35} & a_{36} & b_{33} & a_{37} & a_{38} & b_{34} \\ a_{41} & a_{42} & b_{41} & a_{43} & a_{44} & b_{42} & a_{45} & a_{46} & b_{43} & a_{47} & a_{48} & b_{44} \\ a_{51} & a_{52} & b_{51} & a_{53} & a_{54} & b_{52} & a_{55} & a_{56} & b_{53} & a_{57} & a_{58} & b_{54} \\ a_{61} & a_{62} & b_{61} & a_{63} & a_{64} & b_{62} & a_{65} & a_{66} & b_{63} & a_{67} & a_{68} & b_{64} \\ a_{71} & a_{72} & b_{71} & a_{73} & a_{74} & b_{72} & a_{75} & a_{76} & b_{73} & a_{77} & a_{78} & b_{74} \\ a_{81} & a_{82} & b_{81} & a_{83} & a_{84} & b_{82} & a_{85} & a_{86} & b_{83} & a_{87} & a_{88} & b_{84} \end{array}\right]$$

Each possibility above is derived by inserting the insertion object at a defined place in the input matrix defined by a mask. The insertion object must have the same size as the input matrix on all dimensions except one, called the *dimension of insertion.* This is necessary so that the output matrix will have no missing elements. Therefore, the place of insertion is defined only for one dimension, and the mask definition consists of a pair $(n,(o,l,d))$ where n

is the dimension of insertion, o, l and d are the offset, window size and inter-window spacing on dimension n, respectively. For the list of inserted regions displayed above, the mask definitions are $(2,(4,2,4))$ and $(2,(2,1,2))$.

A spatial inserter operator on matrices must therefore have a defined place to insert an object into the input matrix. Inserter operators are defined in $\mathcal{PGL}$ using the **def-insert** construct:

def-insert *inserter-name* = **make-insert** *window*;

where *window* is the list defined above. This statement binds *inserter-name* to a function with signature $m \times m \rightarrow m$. $\mathcal{PGL}$ contains several built-in spatial inserters; these are listed in a later section.

3.3 Streams of Objects

Streams in $\mathcal{PGL}$ are sequences of basic objects. The basic objects in a stream are separated by a fixed, average amount of time called the *period* of the stream. The period of a stream is defined relative to a base input stream in the application as defined in the high-level system model. If the base input stream is identified in the program, then the period of any stream in the application can be derived by tracing through the program graph and identifying how each function in the graph affects the relative frequency of the stream. Functions which operate on basic objects, *e.g.*, the operators described previously, are *mapped* to object streams. This means that inputs to all functions defined in $\mathcal{PGL}$ are streams and functions are applied to each object in the stream.

$\mathcal{PGL}$ contains a construct for defining input streams from sensors in the application in the program. These are defined as follows:

def-signal *signal-name* = **make-signal** *rel-freq object-rank channel-tag*;

This statement binds *signal-name* to an input stream with relative frequency, size and dimension and channel as given by the parameters to **make-signal**. If the relative frequency is one, then the input stream is a base input stream to the application system as defined in Chapter 2. The *object-rank* identifies the object on the stream; *object-rank* is $(n_1, \ldots, n_r)$ where r is the rank of the matrix and n_i is the size of the matrix on dimension i. The *channel-tag* is a unique identifier of the input stream.

The application pipeline defined by a $\mathcal{PGL}$ program originates at one or more input streams; therefore, we can derive the relative frequency, size and channel information for any stream in the application if we can always predict how a function affects a stream. Therefore, the programmer need only specify this information for the input streams. This is achieved by the **def-signal** construct above.

Signals defined with **def-signal** correspond to actual inputs to the application; there is a similar construct for outputs to the application. Functions in $\mathcal{PGL}$ have only one output; this causes a problem when we have multiple target reports at the top level. $\mathcal{PGL}$ contains a special function to eliminate this problem, the function **multiple-signal-report**. This function is allowed only at the top level application definition, and acts like the **multiple-value-pass** function in Common Lisp [25]. The multiple-signal-report function translates into a special function in the graph which acts as a non-deterministic switch. Since this construct is allowed only at the end of an application, the structure and consistency of the graph are not affected. Future versions of the $\mathcal{PGL}$ may relax this constraint.

3.4 Operations on Streams

Operations on streams fall into two categories: special operations on streams (called temporal sampling operations) and user-defined operations on streams. The latter category is divided into two sub-categories: user-defined operators and procedures. User-defined operators allow a means of describing functions operationally without specifying them mathematically. Procedures are the basic building blocks of the application pipeline and contain special constructs for exploiting parallelism in application functions. By allowing only well-defined $\mathcal{PGL}$ constructs in procedure definitions, the effect of procedures on streams can be determined. The above categories of stream operations are described in the next sections.

3.4.1 Temporal Sampling Operations

$\mathcal{PGL}$ contains operators for a special class of operations on streams called *sampling* operations. Some of these operators, the sampler operators, extract certains values from the stream and discard the rest, yielding a new stream consisting of certain values from the input stream according to a pre-defined pattern. Other operators, the changer operators, replace objects on one stream with objects from another stream in a pre-defined pattern, yielding a new output stream. Other operators, the *placer* operators, insert objects from one stream between objects from another stream in a pre-defined pattern, yielding a new output stream.

Samplers

Sampling of objects from an input stream occurs in regular patterns, therefore we can generalize sampling in one construct. Some of the ways in which we will want to sample objects from a stream are depicted below. If the input stream of objects is

$$[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, \ldots]$$

then the following list shows some of the patterns of sampling which may be desired and the streams that result from the sampling.

$$[a_2, a_3, a_4], [a_0, a_1, a_4, a_5, a_8, a_9, \ldots]$$

Each possibility above is derived by applying a *temporal mask* to the input stream. In order to generalize all the above possibilities, we must identify the components of the mask. Notice that the first stream in the list is of finite length.

The mask in both cases consists of a basic *window*, or shape, an offset from the beginning of the stream and an inter-window spacing for cases where there is an infinite sampling in time. In the first case, the window is three successive objects from the stream, the offset is two, since the window starts with the third object, and the spacing is zero, since there is no infinite sampling. The description of a pattern will therefore consist of the list (o, l, d) where o is the offset of the window from the beginning of the stream, l is the size of the window and d is the inter-window spacing. For the list of streams above, the pattern definitions are as follows: $(2, 3, 0), (0, 2, 2)$, respectively.

A temporal sampler operator on streams of objects must therefore have a defined window to sample from the input stream. Sampler operators are defined in $\mathcal{PGL}$ using the `def-sample` construct:

`def-sample` *sampler-name* = `make-sample` *window;*

where *window* is the list defined above. This statement binds *sampler-name* to a function from one stream to another which leaves each basic object on the stream unaffected. $\mathcal{PGL}$ contains several built-in temporal samplers; these are listed in a later section.

Changers

Changing of objects on a stream occurs in regular patterns, therefore we can generalize "changing" in one construct. Some of the ways in which we will want to change values on a stream are depicted below. If the input stream and new value stream are

$$[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, \ldots], [b_0, b_1, b_2, b_3, b_4, \ldots]$$

respectively, then the following list shows some of the streams which might result from changing.

$$[a_0, b_0, b_1, a_3, a_4, b_2, b_3, a_7, a_8, \ldots], [a_0, a_1, b_0, b_1, b_2, b_3, a_6, a_7, a_8, \ldots]$$

Each possibility above is derived by replacing the values on the input stream with the values on the second stream in a predefined pattern. Patterns for changing the objects on streams are defined in exactly the same manner as patterns for sampling, however, instead of objects to be sampled, the pattern identifies objects to be overwritten. For the list of streams displayed above, the pattern definitions are as follows: $(1,2,2),(2,4,0)$. The zero value for the spacing in the second list indicates that the stream of replacement objects is finite.

A temporal changer operator on streams must therefore have a defined window to change on the input stream. Changer operators are defined in $\mathcal{PGL}$ using the **def-change** construct:

def-change *changer-name* = **make-change** *window;*

where *window* is the list defined above. This statement binds *changer-name* to a function from two streams to one that leaves each basic object on the streams unchanged. $\mathcal{PGL}$ contains several built-in temporal changers; these are listed in a later section.

Placers

Insertion of objects onto a stream occurs in regular patterns and therefore we can generalize placement into one construct. Some of the ways in which we will want to insert values into a stream are depicted below. If the input stream and new value stream are

$$[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, \ldots], [b_0, b_1, b_2, b_3, b_4, \ldots]$$

respectively, then the following list shows some of the streams which might result from changing.

$$[a_0, a_1, b_0, a_2, a_3, b_1, a_4, a_5, b_2, a_6, a_7, b_3, a_8, \ldots], [a_0, a_1, a_2, a_3, a_4, b_0, b_1, b_2, b_3, a_5, a_6, a_7, a_8, \ldots]$$

Each possibility above is derived by inserting the values on the new value stream onto the input stream at a predefined place defined by a mask. Masks for placing objects onto a stream are defined in exactly the same manner as masks for changing and sampling, however, instead of describing values to be sampled or changed, the defined windows of objects are inserted into the stream. For the list of possibilities displayed above, the mask definitions are as follows: (2,1,2),(5,4,0). The zero value for the spacing in the second list indicates that the stream of insertion objects is finite.

A temporal placer operator on streams must therefore have a defined window to insert on the input stream. Placer operators are defined in $\mathcal{PGL}$ using the **def-place** construct:

> **def-place** *placer-name* = **make-place** *window;*

where *window* is the list defined above. This statement binds *placer-name* to a function from two streams to one which leaves each basic object on the streams unchanged. $\mathcal{PGL}$ contains several built-in temporal placers; these are listed in a later section.

3.4.2 User-Defined "Primitive" Operators

$\mathcal{PGL}$ contains a construct for the programmer to define operators on object streams which behave like primitives in the language. These are low-level functions on object streams which the programmer may not wish to code in $\mathcal{PGL}$, *i.e.*, he may wish only to describe these functions in terms of their operational characteristics. This has the effect of designating some user-determined operations as non-partitionable and therefore gives the programmer a means to control granularity if he so desires. These functions must be included in the program graph as operators which are defined in $\mathcal{PGL}$ with the following special construct:

> **def-op** *op-name* = **make-op** *body-op-count init-op-count memory-size-count input-type output-type frequency-factor*;

where the parameters reflect some user-defined measure of the computational latency of the operator which is consistent with the rest of the application, *i.e.*, if latency is measured in FLOPS, then *body-op-count* and *init-op-count* must be the number of FLOPS in *op-name*. The *init-op-count* is the initial processing overhead for the operator; the *body-op-count* is the average computational latency of the operator. The *input-type* and *output-type* parameters specify the input and output objects, respectively. These are lists of the form $(n_1, \ldots, n_r)$ where r is the rank of the matrix and n_i is the size of the matrix on dimension i. The *memory-size-count* is some user-defined measure of the memory used by the operator. The *frequency-factor* is the factor by which the relative frequency of the input stream is multiplied to obtain the frequency of the output stream.

3.5 Procedures

A $\mathcal{PGL}$ procedure is a user-defined transformation from one or more streams of basic objects to another. The programmer describes the functionality of the procedure in terms of the $\mathcal{PGL}$ constructs described previously and new constructs described in the next sections. By limiting the ways in which procedures can be defined we can insure that we can always derive the characteristics of the output stream of any procedure.

Procedures are defined in $\mathcal{PGL}$ as follows:

`def` *function-name arg1 arg2 ... argn = body-statement;*

where the *args* are the formals of the procedure. This statement binds *function-name* to a function of arity *n* with body *body-statement.* The *body-statement* is either a procedure or operator invocation, a block statement or an iteration.

3.5.1 Block Statements

Block statements are provided in $\mathcal{PGL}$ to facilitate the definition of operators and procedures. Block statements can be used in any `def-` construct described previously in this chapter as long as the semantics and signature of the operator definitions are kept intact. The block statement consists of a set of bindings and an expression to be evaluated in the environment produced by the bindings. The environment of the block statement will determine the data dependencies within the procedure and the structure of the program graph. Block constructs in $\mathcal{PGL}$ are written as in Id:

```
{statement;
 ...
 statement
 in
   expression}
```

Each statement in the block is an identifier binding which determines the data dependencies in the result expression. The expression is either a value, a procedure or operator invocation, or an iteration.

3.6 Iterations

Iterations are loop-like constructs which denote repetitive processing within a function. There are two types of iterations: *spatial* and *temporal*. Spatial iteration refers to repetitive processing of a matrix in which the same function is performed on regularly selected regions from the matrix. These regions exhibit the same characteristics as the regions selected by a spatial selection construct and will be defined in the same way. Temporal iteration refers to repetitive processing over groups of objects selected from an objects stream. These groups of objects exhibit the same characteristics as the groups sampled by a temporal sampling construct and will be defined in the same way.

3.6.1 Spatial Iteration

Spatial iteration is performed on each basic object (matrix) from a stream. A spatial iteration is a loop over different regions within the matrix. We will therefore need a means to define both the shape of a region and the collection of regions over which the function will loop. The regions are defined in the same manner as the windows in spatial selection constructs; therefore, it remains only to define how these regions are chosen. Some of the lists of regions over which a function can iterate are depicted below. If the input matrix is

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix}$$

then the following are lists of selected windows over which the function can iterate:

$$\left(\begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \\ a_{42} \\ a_{52} \\ a_{62} \\ a_{72} \\ a_{82} \end{bmatrix}, \begin{bmatrix} a_{14} \\ a_{24} \\ a_{34} \\ a_{44} \\ a_{54} \\ a_{64} \\ a_{74} \\ a_{84} \end{bmatrix}, \begin{bmatrix} a_{16} \\ a_{26} \\ a_{36} \\ a_{46} \\ a_{56} \\ a_{66} \\ a_{76} \\ a_{86} \end{bmatrix}, \begin{bmatrix} a_{18} \\ a_{28} \\ a_{38} \\ a_{48} \\ a_{58} \\ a_{68} \\ a_{78} \\ a_{88} \end{bmatrix}\right).$$

$$\left(\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \begin{bmatrix} a_{15} & a_{16} \\ a_{25} & a_{26} \end{bmatrix}, \begin{bmatrix} a_{51} & a_{52} \\ a_{61} & a_{62} \end{bmatrix}, \begin{bmatrix} a_{55} & a_{56} \\ a_{65} & a_{66} \end{bmatrix}\right)$$

$$(\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \end{bmatrix}, \begin{bmatrix} a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \end{bmatrix},$$

$$\begin{bmatrix} a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix}, \begin{bmatrix} a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \\ a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \end{bmatrix})$$

Each list above is generated by applying a spatial mask to the input matrix at selected points along each dimension. The mask consists of a basic shape, or window, and an inter-window spacing on each dimension. The selected points are defined by an offset and an inter-region spacing on each dimension. In the first case, the window is one column (1×8), the inter-window spacing is zero, *i.e.*, none, the offsets of the defined points are 0 and 1 on dimensions 1 and 2, respectively, and the inter-region spacings are 0 and 1 on dimensions 1 and 2, respectively. In generating the lists, the matrix is traversed in a particular order; this is demonstrated in the second case, where the 4×4 matrices are selected along the row and then along the column, *i.e.*, in row-major order. The list of regions must also be characterized by an order of traversal.

The description of the list of regions will therefore consist of three lists w, p, and o. The list w describes the basic window; $w = ((l_1, d_1), \ldots, (l_r, d_r))$ where r is the rank of the matrix, l_i is the size of the window and d_i is the inter-window spacing on dimension i. The list p defines the points at which the window is selected for the iteration; $p = ((o_1, s_1, \ldots, o_r, s_r))$ where r is the rank of the matrix, o_i is the offset and s_i is the inter-region spacing on dimension i. The list o defines the order in which the matrix is traversed; $o = (n_1, \ldots, n_r)$ where n_i is ith dimension to be traversed.

For the three cases displayed above, the iteration parameters are as follows:

1. $w = ((8,0),(1,0)), p = ((0,0),(1,1)), o = (1,2)$
2. $w = ((2,0),(2,0)), p = ((0,3),(0,3)), o = (2,1)$
3. $w = ((2,3),(8,0)), p = ((0,0),(0,0)), o = (1,2)$

A spatial iterator must therefore have a defined window, point-list and order to iterate over regions of the input matrix. Spatial iterators are defined in $\mathcal{PGL}$ as follows:

`def-iter` *iter-name* = `make-iter` *window point-list order*;

where *window, point-list,* and *order* are lists as defined above. This statement binds *iter-name* to a function from a matrix to a list of regions within the matrix. Generation of the regions is determined in the definition of *iter-name*.

Spatial iterations in $\mathcal{PGL}$ are written as follows:

`generate` *identifier-0, ... , identifier-n* `from` *region-list-0, ... , region-list-n* `in`
body-statement
`compose` *composer*;

where each *region-list* is generated by applying a spatial iterator to a matrix. The spatial iteration construct is interpreted as follows: for each *region-i* described in *region-list-i*, *identifier-i* is bound to *region-i*. The *body-statement* is then evaluated for each binding, producing a list of result objects. The body-statement is either a procedure or operator application or another spatial iteration. The list of result objects is combined into one result object by applying the *composer* to the list of result objects. A composer is defined by the same parameters as the iterator, however, instead of defining a list of regions to be selected from a matrix, the composer parameters define the regions of an output matrix into which each object in the result object list is transformed. Spatial composers are defined in $\mathcal{PGL}$ as follows:

`def-comp` *comp-name* = `make-comp` *window point-list order*;

where *window, point-list,* and *order* are lists as defined above. This statement binds *comp-name* to a function from a list of regions within a matrix to a matrix. Composition of the regions is determined in the definition of *comp-name*.

3.6.2 Temporal Iteration

Temporal iteration is performed on a group of objects from a stream of objects. A temporal iteration is a loop over different groups sampled from a stream. We will therefore need a means to define both the size of the group from the stream and the collection of groups over which the function will loop. The groups are defined in the same manner as the groups in the temporal sampling constructs; therefore, it remains only to define how these groups are collected from the stream. Some of the groups of objects over which the function can iterate are depicted below. If the input stream of objects is

$$[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, \ldots]$$

then the following are lists of grouped streams over which the function may iterate.

$$([a_1, a_2, a_3, a_4],\ \ [a_4, a_5, a_6, a_7], \ldots)$$

$$([a_0, a_1, a_4, a_5, a_8, a_9, \ldots],\ \ [a_2, a_3, a_6, a_7, \ldots])$$

The above lists are lists of *grouped streams.* Grouped streams are streams in which groups of values are viewed as a unit for processing in one function invocation. In the first case, the list is infinite, but each grouped stream is a finite stream composed of one group. In the second case, the list is finite, but each grouped stream is an infinite sequence of finite groups. A non-grouped stream can be thought of as an infinite grouped stream with group size equal to one.

Each list above is generated by applying a temporal mask to the input stream at selected points in the stream. The mask consists of a basic *window*, or shape, and an inter-window spacing for cases where there is an infinite sampling in time. The selected windows are defined by an offset from the beginning of the stream and an inter-window spacing in the stream. In the first case, the window is four successive objects from the stream and the inter-window spacing is zero, *i.e.*, none. The offset from the beginning of the stream is one and the inter-window spacing is three.

The description of a list of grouped streams will therefore consist of two lists w and p. The list w describes the mask; $w = (l, d)$ where l is the size of the window and d is the inter-window spacing. The list p defines the points in the stream at which the window is defined for the iteration; $p = (o, s)$ where o is the offset from the beginning of the stream and s is the inter-mask spacing. For the two cases displayed above, the iteration parameters are as follows: $w = (4, 0), p = (1, 3)$ and $w = (2, 2), p = (0, 1)$, respectively.

A temporal iterator must therefore have a defined window and point-list in order to iterate over groups from the input stream. Temporal iterators are defined in $\mathcal{PGL}$ as follows:

```
def-repeat iter-name = make-repeat window point-list;
```

where *window* and *point-list* are lists as defined above. This statement binds *iter-name* to a function from a stream of matrices to a list of grouped streams of matrices. Generation of the grouped streams is determined in the definition of *iter-name*.

Temporal iterations in $\mathcal{PGL}$ are written as follows:

```
repeat identifier-0, ... , identifier-n over group-list-0, ... , group-list-n in
tfold fold-function identifier-0 ... identifier-k init identifier-k+1 ... identifier-n
collect collector;
```

where each *group-list* is generated by applying a temporal iterator to a stream of matrices. The temporal iteration construct is interpreted as follows: for each *group-i* described in *group-list-i*, *identifier-i* is bound to the grouped stream *group-i*. The special function `tfold` takes a function to fold over a group and the grouped streams to which the function is applied. One of the function inputs is given the keyword `init` instead of a grouped stream; this input is designated as the feedback. This input value is initialized at the beginning of each group and given the results of the previous invocation of the function during the processing of the group. The inital value is specified by binding `init` to the initial value. If the group is composed of windows with an inter-window spacing, *i.e.*, $d \neq 0$, then the designated feedback input is initialized only in the first function invocation. The function is folded over each group and outputs an object (a group of size one) for each input group. The *fold-function* must be an operation on basic objects, *i.e.*, no nesting of temporal iterations is allowed in $\mathcal{PGL}$.

The result of the `tfold` is a list of grouped streams, all of which have group size one. One result stream is obtained by applying the *collector* to this list of grouped streams. A collector is defined by the same parameters as the iterator, however, instead of defining a list of grouped streams to be derived from an input stream, the collector parameters define the groups of an output stream into which each group in the result stream list is transformed. Temporal collectors are defined in $\mathcal{PGL}$ as follows.

```
def-collect comp-name = make-collect window point-list;
```

where *window* and *point-list* are lists as defined above. This statement binds *comp-name* to a function from a list of grouped streams with group size one to a stream of objects. Collection of the groups is determined in the definition of *comp-name*.

The lists used in all iteration constructs and also in the parameter definitions for spatial and temporal iteration and selection constructs are internal data structures used only in these constructs, and therefore are not included in the definition of objects.

3.7 Library Functions

Examples of the above $\mathcal{PGL}$ constructs can be found in Appendix C, which presents the $\mathcal{PGL}$ code for the radar appplication. $\mathcal{PGL}$ also contains library functions which are built-in spatial and temporal iterators, composers and selectors. The user creates his own functions using the `make-` functions provided in $\mathcal{PGL}$. The following are frequently-used functions which can be defined with the `make-` constructs but are included in $\mathcal{PGL}$ for convenience.

Spatial Iterators:
`gen-vect-elements` iterates over the elements of a vector
`gen-matrix-elements` iterates over the elements of a matrix (row-major)
`gen-matrix-columns` iterates over the columns of a matrix
`gen-matrix-rows` iterates over the rows of a matrix
Spatial Selectors:
`select-column-n` selects the nth column from a matrix
`select-row-n` selects the nth row from a matrix
`n-columns-o` selects *n* columns at offset *o*
`n-rows-o` selects *n* rows at offset *o*
`sample-n-columns-every-m` selects columns periodically
`sample-n-rows-every-m` selects rows periodically
Spatial Composers:
`comp-vector-elements` composes a vector by elements
`comp-vector-elements-back` composes a vector by elements backwards
`comp-matrix-elements` composes a matrix by elements (row-major)
`comp-matrix-elements-back` composes a matrix by elements (col-major)
`comp-matrix-columns` composes a matrix by columns
`comp-matrix-rows` composes a matrix by rows
`comp-matrix-columns-back` composes a matrix by columns backwards
`comp-matrix-rows-back` composes a matrix by rows backwards
Spatial Replacers:
`replace-column-n` replaces the nth column in the matrix
`replace-row-n` replaces the nth row in the matrix
`replace-vect-element-n` replaces the nth vector element
`replace-matr-element-n-m` replaces the matrix element at (n, m)
Spatial Inserters:

insert-column-n inserts after the nth column in the matrix
insert-row-n inserts after the nth row in the matrix
insert-vect-element-n inserts after the nth vector element
Temporal Iterators:
m-every-n iterates over groups of size m every n stream values
Temporal Selectors:
every-nth selects every nth value from the stream
Temporal Composers:
compose-m-n composes groups of size m every n stream values
Temporal Replacers:
replace-stream-n replaces every nth value from the stream
Temporal Inserters:
insert-stream-n inserts after every nth value from the stream

Chapter 4

Program Graph Representation

The program graph representation described in this chapter is the target language for compilation of an application program written in the language $\mathcal{PGL}$. The program graph consists of nodes and arcs where arcs represent *signals* and nodes represent either

- signal processing functions, or
- data routing operators.

Function nodes in the program graph are represented by rectangles while data routing operators are represented by the special shapes defined later in this chapter. Signals and nodes contain special parameters which are used by the strategy module of the compiler to find an optimal partition on a given hardware configuration. Some of these parameters to signals, functions, and data routing operators are fixed at compile time; others are variable and are used in evaluation of the partition during the decomposition phase. In this chapter, after defining signals, we will describe all functions and data routing operators and their output signals.

4.1 Signals

Signals are sets of parallel, disjoint streams which can vary in size. Each stream in a signal has the same frequency relative to the base input stream of the application. The tokens on each stream in a signal have the same shape and size. Each stream in a signal may originate at a different sensor input stream, since different lines of processing can be combined into one signal.

Signals are characterized by four parameters reflecting the above information. A signal is therefore a tuple $< w, d, f, c >$ where:

- w is the width of the signal, *i.e.*, the number of parallel streams represented by the arc;
- d specifies the dimension and size of the basic object on each stream in the signal;
- f is the relative frequency of each stream in the signal specified as a multiple of the base input stream;
- c is the channel designator of the signal, which identifies the input sources for the streams in the signal.

These values are explained in more detail in the next paragraphs. Signals will be denoted in this work by the notation $sig(w, d, f, c)$.

Stream Width: Each arc in the program graph represents a signal which is either:

- a stream of objects, called *tokens*, or,
- a set of streams of objects in parallel.

In order to generalize signals to include both possibilities, the *width* w of the arc in the graph is included in the signal concept. w is an integer specifying the number of parallel streams actually present on the given arc. Stream width is a parameter which is set by an operator and determines the level of partitioning. Since each arc is an input to the function, the width of that signal determines how many parallel, independent tasks are implemented to perform the function. The type of operator used to manipulate the input stream depends on which of the four types of partitioning is in place for that function.

Dimension and Size: Each token on every stream in the signal is a basic object with rank r. Since basic objects are finite, each basic object has a defined size on every dimension. The dimension and size of each token is represented as a list of integers $(n_1, n_2, \ldots, n_r)$ where n_i is the size of the matrix on dimension i.

Dimension and size information is defined for each individual stream of values actually represented by an arc. If an arc has stream width w, then it represents w parallel streams where each of the w streams has tokens as described by the dimension and size information.

Relative Frequency: Each stream in the graph has an average frequency of token flow. Program graph representation is concerned only with the frequency of a given stream *as it is defined relative to the base input stream.* Each arc is characterized by a *relative frequency* $F_r = C_f \cdot F_s$ where F_s is the global frequency of the sensor input. The value f in the signal tuple defines the factor C_f. C_f can be less than, greater than or equal to one, since the frequency of any stream is unrestricted relative to the base input stream.

Relative frequency is defined for each individual stream of values actually represented by an arc. If an arc has stream width w, then it represents w parallel streams where each of the w streams has relative frequency as specified for the given arc.

Channel: Each token is associated with the processing of a different *channel* in the application. A stream is tagged with channel information for cases where channels must be kept distinct if processed in the same task. The channel tag is analogous to the event number tag, except instead of keeping different tokens in time separated, this tag keeps different tokens in space separated. This is necessary because some of the signal processing functions are history-sensitive. The channel tag insures that the correct history is being used.

The channel designator identifies in a uniform way the mapping of streams in the signal to input channels. Since a stream or signal may also be composed of data originating from different channels, the channel designator representation must be recursive.

The value c in the signal description must represent accurate channel information for each stream in the signal. Each stream may itself be composed of data from different channels. The channel descriptor representation is therefore a composite, recursive data structure; it is composed of w items which are also channel descriptors. The representation must be uniform so that the channel descriptor for any stream in the set can be recovered accurately.

The channel descriptor will be represented as a disjoint union which is either a single channel descriptor, *e.g.*, a string or number or other such unique identifier, or a list of channel descriptors of length w. In order to recover channel information accurately, there must be some *a priori* order on the list, *i.e.*, some well-defined order on the streams in the signal corresponding to the list ordering. The actual ordering is arbitrary; any will work as long as consistency is maintained in all graph elements.

A sample ordering could be constructed in the following manner: the channel information for a signal of size w is represented as a list of channel tags. Each channel tag is either another list of channel tags or a unique identifier, *e.g.*, a number or character. The list is of length w, with a channel tag for each stream in the set. Then, if we wanted to split the signal into m sets of streams, we would divide the list into sub-lists of length k/m. To combine several signals into one signal, we simply append the channel tag lists to get one big list for the resulting signal. This assumes that the lists in a channel identifier are *disjoint*, *i.e.*, there are no repeated tags. This can be assured by constructing the transformation rules appropriately; these rules are presented in Chapter 5.

4.2 Functions

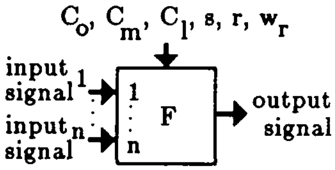
Functions are defined by nodes in the program graph. Functions in the context of the program graph are defined as transformations from one basic object to another and are mapped to input stream values as they arrive. Functions nodes in the graph denote the base-level functions defined in Chapter 2, *i.e.*, the lower-level functions which are applied repetitively to windows of a matrix or stream. These base-level functions are implemented in a number of parallel tasks. The number of tasks in which the base-level function is implemented is determined by the width of the input signal to the function.

Since the width of this signal is variable, the number of tasks is variable, as is the size of the section or the frequency of the input streams to each task. Since the size of the base-function input is constant, function nodes contain no variable parameters. While evaluating a partition, the compiler uses the input size information for base-level functions in the graph and the size of the basic object on the input signal to determine how many times the base-level function is applied in each task.

Functions are characterized by their *cost* in the program graph, which is dependent upon computational latency, memory requirements and overhead processing. This cost is used during decomposition to evaluate a given partition. Since function nodes are base-level functions, no further partitioning is possible and cost parameters must either be derived by the compiler or supplied by the programmer. The following paragraphs describe the graph construct for base-level functions in detail.

Function F $Co\ Cm\ Cl\ s\ r\ w_r$

$$\mathrm{sig}(n_1, f, w, c) \times \ldots \times \mathrm{sig}(n_n, f, w, c)$$
$$\rightarrow \mathrm{sig}'(n', f', w', c')$$



F is the name of the function. C_o is an overhead cost associated with task invocation. C_m is the memory cost of the function and C_l is the computational latency defined relative to some base computation, *e.g.*, integer, real or complex operations. $s = (s_1, \ldots, s_n)$, where n is the number of inputs to the function and each s_i is a list $(m_1, \ldots, m_p)$, where p is the rank of input i and m_j is the size of input i on dimension j. $r = (r_1, \ldots, r_p)$, where p is the rank of the output and r_i is the size of the output on dimension i. The window ratio w_r indicates the ratio of input signal frequency to output signal frequency, which is the size of the function state. For most functions, this value is one.

$n' = (n'_1, n'_2, \ldots, n'_n)$, where $n'_i = g(n_i, s, r, o)$, where o is the overlap parameter of the nearest enclosing operator. Overlap is explained in the later sections on data routing operators. g is a function which derives the number of times the function is applied to the input data and calculates the size of the output section.

$f' = w_r \cdot f$

$w' = w$

$c' = c.$

4.3 Data Routing Operators

Data routing operators are denoted by the special shapes defined in the next sections. These data routing operators describe the partitioning mechanisms described in Chapter 2. Each type of partitioning has a class of operators; these are the following three categories:

- Spatial Operators,
- Temporal Operators, and
- Channel Operators.

corresponding to spatial partitioning, temporal partitioning and independent channel processing, respectively. Consistency is maintained by inserting partitioning operators into the graph in pairs. The first operator in the pair splits each input stream in the input signal into some number of streams which determines the number of tasks in the partition. The second operator in the pair combines the output streams of the tasks in the partition into one output signal.

Since the input to the operator is a signal, the presence of an operator in the graph denotes a *composite operator*, which is a set of *basic operators* in parallel. This is analogous to the manner in which functions in the graph denote sets of parallel function instances. The basic operators operate on individual streams of data; the number of basic operators in parallel determines how the composite operator acts on the entire signal. This allows for the nesting of certain operators, *i.e.*, partitioning each function inside a partitioned function.

The cost of operators, *i.e.*, the sum of latency and memory requirements, is constant and fixed in the evaluation phase. In order to simplify partitioning, operator cost is assumed negligible compared to the cost of functions in the graph. Later, more complex implementations can relax this assumption. The next sections describe the composite data-routing operators.

4.3.1 Spatial Operators

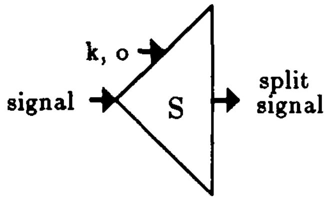
The class of spatial operators contains five sub-classes. The spatial iterating operators split each basic object into sections or repeat each basic object on several parallel streams and form the appropriate output signal. Any of these operators correspond to the first function in the routing function pair defined in Chapter 2. The spatial composing operators combine several parallel streams of basic objects into one. Any of these operators correspond to the second function in the routing pair defined in Chapter 2.

The spatial selection, replacement and insertion operators select, replace and insert, respectively, designated regions of each basic object. The spatial operators operate only on each basic object in a stream and therefore do not change the frequency and channel information for the signal.

Spatial Iterating and Duplicating Operators

The spatial iterating operators split each basic object on each stream in the input signal into sections as determined by the parameters to the operator. Each section forms a basic object on an output stream in the output signal. The spatial duplicating operator duplicates each basic object on each stream in the input signal; each duplicated object is put on an output stream in the output signal. Each of these operators expands the input signal into a signal of greater width and same-sized or smaller tokens. The **Split** operator exploits spatial locality of referency in forming new objects from an input matrix. The **Divide** operator exploits spatial periodicity of reference.

Split k o



$\text{sig}(n, f, w, c) \rightarrow \text{sig}'(n', f', w', c')$

The number of sections is given by $k = (k_1, \ldots, k_r)$, where r is the rank of the input matrix and k_i is the number of sections on dimension i. The overlap between sections is given by $o = (o_1, \ldots, o_r)$, where o_i is the overlap on dimension i.

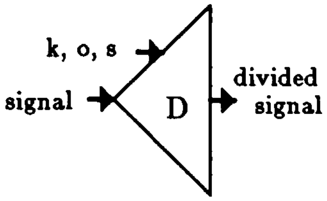
$n' = (n'_1, \ldots, n'_r)$, where $n'_i = n_i/k_i + 2 \cdot o_i$

$f' = f$

$w' = w \cdot \prod_{i=1}^{n} k_i$

$c' = c.$

Divide k o s



$\text{sig}(n, f, w, c) \rightarrow \text{sig}'(n', f', w', c')$

The number of sections is given by $k = (k_1, \ldots, k_r)$, where r is the rank of the input matrix and k_i is the number of sections on dimension i. The overlap between sections is given by $o = (o_1, \ldots, o_r)$, where o_i is the overlap of windows in dimension i. The inter-window spacing is given by $s = (s_1, \ldots, s_r)$, where s_i is the spacing on dimension i. Figure 4.1 shows the definition of these parameters on a sample matrix.

$n' = (n'_1, n'_2, \ldots, n'_n)$, where $n'_i = (s_i/k_i + 2o_i) \cdot n_i/s_i$

$f' = f$

$w' = w \cdot \prod_{i=1}^{i=n} k_i$

$c' = c.$

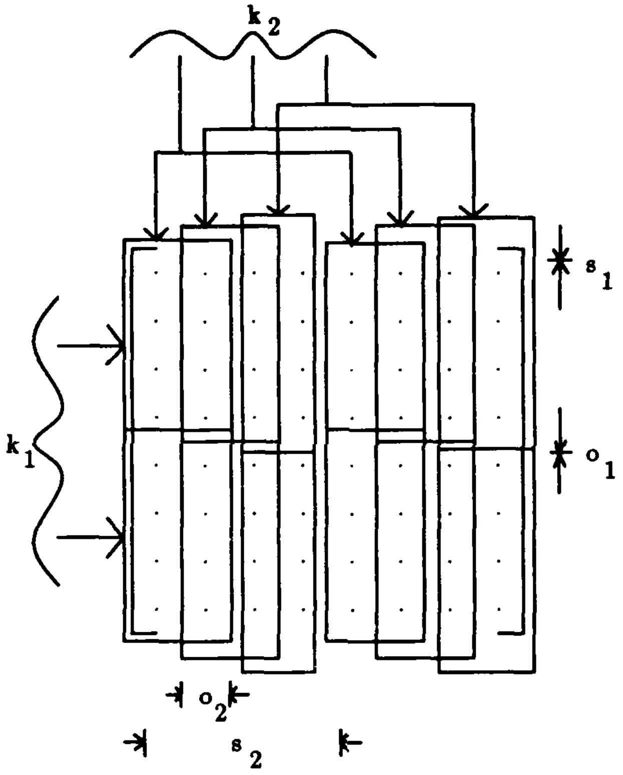
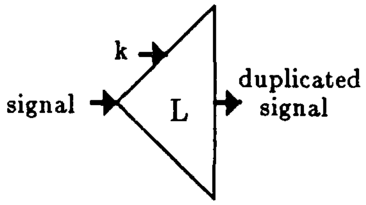


Figure 4.1: Parameters to Divide Operator

Duplicate *k*



$\mathrm{sig}(d, f, w, c) \rightarrow \mathrm{sig}'(d', f', w', c')$

The number of streams upon which each basic object on each stream is duplicated is given by integer k.

$n' = n$

$f' = f$

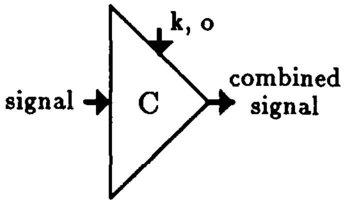
$w' = w \cdot k$

$c' = c.$

Spatial Composing Operators

The spatial composing operators combine the basic objects on all streams in the input signal into another basic object as determined by the parameters to the operator. The new basic object is output to each stream in the output signal. Each of these operators reduces the input signal into a signal of lesser width with larger tokens. The frequency and channel information is unchanged. The `Combine` operator exploits spatial locality of reference; the `Recompose` operator exploits spatial periodicity of reference.

Combine *k o*



$\mathrm{sig}(n, f, w, c) \rightarrow \mathrm{sig}'(n', f', w', c')$

The number of basic objects to combine is given by $k = (k_1, \ldots, k_r)$, where r is the rank of the output matrix and k_i is the number of sections on dimension i. The overlap between sections is given by $o = (o_1, \ldots, o_r)$, where o_i is the overlap on dimension i.

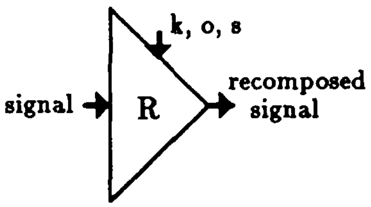
$n' = (n'_1, n'_2, \ldots, n'_n)$, where $n'_i = (n_i - 2o_i) \cdot k_i$

$f' = f$

$w' = \frac{w}{\prod_{i=1}^{n} k_i}$

$c' = c.$

Recompose $k\ o\ s$



$$\mathrm{sig}(n,f,w,c) \rightarrow \mathrm{sig}'(n',f',w',c')$$

The number of sections is given by $k = (k_1, \ldots, k_r)$, where r is the rank of the input matrix and k_i is the number of divisions on dimension i. The overlap between windows is given by $o = (o_1, \ldots, o_r)$, where o_i is the overlap on dimension i. The inter-window spacing is given by $s = (s_1, \ldots, s_r)$, where s_i is the spacing on dimension i. These parameters have the same definition as the parameters to the Divide operator.

$n' = (n'_1, n'_2, \ldots, n'_n)$, where $n'_i = n_i s_i / (\frac{s_i}{k_i} + 2o_i)$

$f' = f$

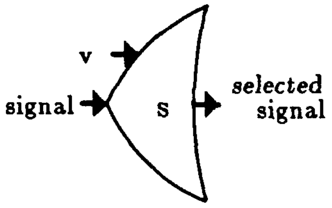
$w' = \frac{w}{\prod_{i=1}^{i=n} k_i}$

$c' = c.$

Spatial Selection Operators

The spatial selection operators select portions of each input matrix on each stream in the input signal as determined by the parameters. The portion of the matrix selected forms a basic object output to each stream in the output signal. These operators preserve width and change only token information. The Select operator exploits spatial locality of reference; the Choose operator exploits spatial periodicity of reference.

Select v



$$\mathrm{sig}(n,f,w,c) \rightarrow \mathrm{sig}'(n',f',w',c')$$

The region of the matrix selected is given by $v = ((m_1,k_1), \ldots, (m_r,k_r))$, where r is the rank of the input matrix, m_i is the offset of the window selected and k_i is the size of the window selected on dimension i.

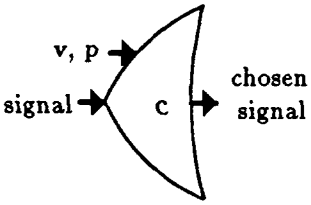
$n' = (k_1, k_2, \ldots, k_n)$

$f' = f$

$w' = w$

$c' = c.$

Choose v p



$\mathrm{sig}(n, f, w, c) \rightarrow \mathrm{sig}'(n', f', w', c')$

The windows of the matrix selected are given by $v = ((m_1, k_1), \ldots, (m_r, k_r))$, where r is the rank of the input matrix, m_i is the offset of the first window and k_i is the size of ea^h window on dimension i. The inter-window spacing is given by $p = (p_1, \ldots, p_r)$, where p_i is the spacing on dimension i.

$n' = (n'_1, n'_2, \ldots, n'_n)$, where $n'_i = n_i k_i / p_i$

$f' = f$

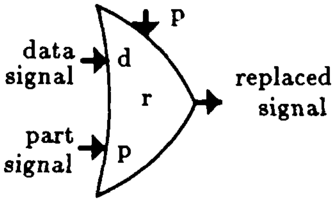
$w' = w$

$c' = c.$

Spatial Replacement Operators

The spatial replacement operators replace portions of each input data matrix on each stream in the input data signal with the input part matrix as determined by the parameters. The new matrix with defined portions replaced is output to each stream in the output signal. These operators preserve width and change only token information. The `Replace` ,perator exploits spatial locality of reference; the `Substitute` operator exploits spatial periodicity of reference.

Replace p



$\mathrm{sig}(n_d, f_d, w_d, c_d) \times \mathrm{sig}(n_p, f_p, w_p, c_p)$
$\rightarrow \mathrm{sig}'(n', f', w', c')$

The place at which the part matrix is substituted for the elements of the data matrix is $p = (p_1, \ldots, p_r)$, where r is the rank of the data matrix and p_i is the offset of replacement on dimension i.

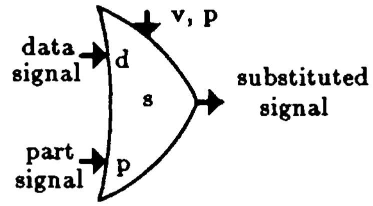
$n' = n_d$

$f' = f_d$

$w' = w_d$

$c' = c_d.$

Substitute v p



$\text{sig}(n_d, f_d, w_d, c_d) \times \text{sig}(n_p, f_p, w_p, c_p)$
$\rightarrow \text{sig}'(n', f', w', c')$

The windows of the data matrix which are replaced by the part matrix are defined by $v = ((m_1, k_1), \ldots, (m_r, k_r))$ and $p = (p_1, \ldots, p_r)$, where r is the rank of the data matrix, m_i is the offset, k_i is the window size and p_i is the inter-window spacing on dimension i.

$n' = (n_{d1}, n_{d2}, \ldots, n_n)$

$f' = f$

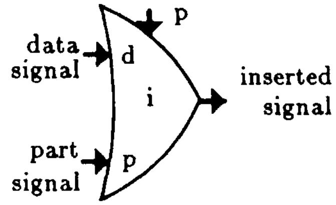
$w' = w$

$c' = c.$

Spatial Insertion Operators

The spatial insertion operators insert windows into each data matrix on each stream in the input data signal with input part matrix as determined by the parameters. The new matrix with new portions inserted is output to each stream in the output signal. These operators preserve width and change only token information. The `Insert` operator exploits spatial locality of reference; the `Append` operator exploits spatial periodicity of reference.

Insert p



$\text{sig}(n_d, f_d, w_d, c_d) \times \text{sig}(n_p, f_p, w_p, c_p)$
$\rightarrow \text{sig}'(n', f', w', c')$

The place at which the part matrix is inserted into the data matrix is given by $p = (r, s)$, where r is the dimension of the insertion and s is the offset of insertion on dimension r.

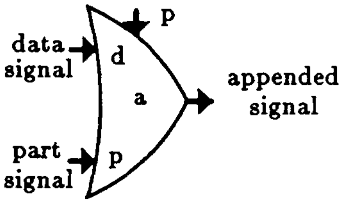
$n' = (n'_i, n'_2, \ldots, n'_n)$, where $n'_i = n_{di} + n_{pi}$

$f' = f_d$

$w' = w_d$

$c' = c_d$.

Append p



$\text{sig}(n_d, f_d, w_d, c_d) \times \text{sig}(n_p, f_p, w_p, c_p)$
$\rightarrow \text{sig}'(n', f', w', c')$

The places at which the part windows are inserted into the data matrix is given by $p = (r, m, k, s)$, where r is the dimension of insertion, m is the offset, k is the window size and s is the inter-window spacing on dimension r.

$n' = (n'_i, n'_2, \ldots, n'_n)$, where $n'_i = n_{di} + n_{pi}$

$f' = f_d$

$w' = w_d$

$c' = c_d$.

4.3.2 Temporal Operators

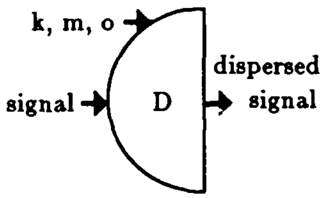
The class of temporal operators contains five sub-classes. The temporal iterating operators split each stream into several parallel streams or repeat each basic object several times on several parallel streams by distributing tokens. Any of these operators corresponds to the first function in the routing function pair defined in Chapter 2. The temporal composing operators combine several parallel streams into one. Any of these operators corresponds to the second function in the routing function pair defined in Chapter 2.

The temporal sampling, changing and placing operators select, replace and insert, respectively, designated windows of each input stream. The temporal operators operate only on the tokens on a stream and therefore do not change the size and dimension information for the objects in a signal.

Temporal Iterating and Repeating Operators

The temporal iterating operators split each stream in the input signal into several grouped streams. Each grouped stream in the output signal is processed in a special graph construct corresponding to the grouped stream, therefore, the output signal does not reflect the grouping information. The temporal repeating operator outputs a group of values on each stream in the output signal, where each token in the group is a duplicate of the corresponding token on the input stream. Each of these operators expands the input signal into a signal of greater width. The iterating operators reduce the frequency of each stream; the repeating operator increases the frequency. The `Disperse` operator exploits temporal locality of reference; the `Rotate` operator exploits temporal periodicity of reference.

Disperse $k\ m\ o$



$\mathrm{sig}(n, f, w, c) \rightarrow \mathrm{sig}'(n', f', w', c')$

The number of grouped streams for each stream in the input signal is k, the size of each group (temporal window) in each output stream is m and the overlap between successive groups is o.

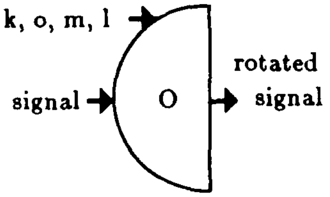
$n' = n$

$f' = f/k + 2o$

$w' = w \cdot k$

$c' = c.$

Rotate $k\ o\ m\ l$



$\mathrm{sig}(n, f, w, c) \rightarrow \mathrm{sig}'(n', f', w', c')$

The number of grouped streams for each stream in the input signal is k, the size of each group in each output stream is m, the overlap between successive groups is o and the inter-window spacing is l. Figure 4.2 shows the definition of these parameters on a sample stream.

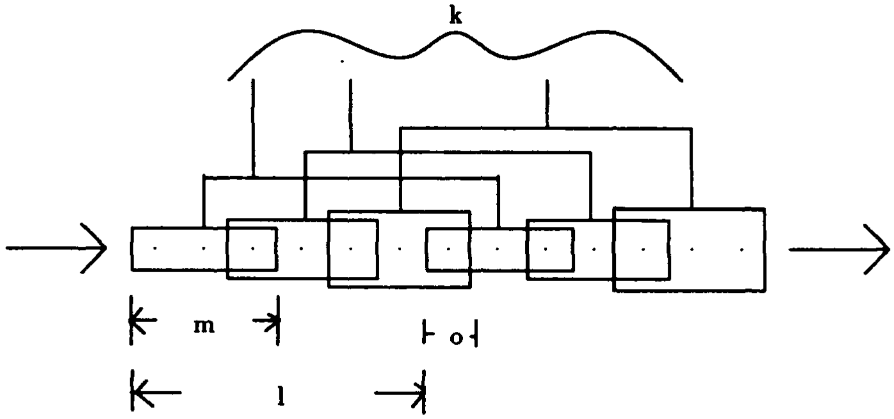


Figure 4.2: Parameters to Rotate Operator

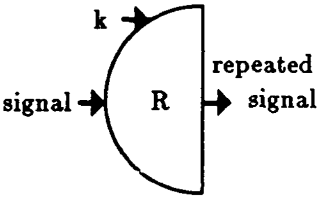
$n' = n$

$f' = f/k + 2mo/l$

$w' = w \cdot k$

$c' = c.$

Repeat k m



$\mathrm{sig}(n, f, w, c) \rightarrow \mathrm{sig}'(n', f', w', c')$

The number of grouped streams for each stream in the input signal is k, the size of the group into which each token on each stream in the input signal is expanded is m.

$n' = n$

$f' = f \cdot m$

$w' = w \cdot k$

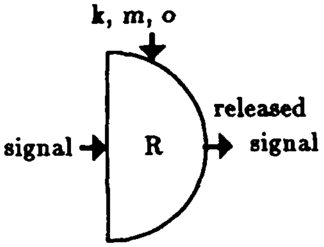
$c' = c.$

Temporal Composing Operators

The temporal composing iterators combine the grouped streams in the input signal into another signal, where the tokens from the input streams have been merged into one stream according to the parameters. These operators reduce the width of the input signal and raise the frequency.

The Release operator exploits temporal locality of reference; the Collect operator exploits temporal periodicity of reference.

Release $k\ m\ o$



$\mathrm{sig}(n, f, w, c) \rightarrow \mathrm{sig}'(n', f', w', c')$

The number of grouped streams to merge is given by k, the size of the groups (temporal windows) in each grouped stream is given by m and the overlap between the successive windows is given by o.

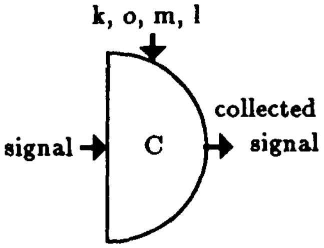
$n' = n$

$f' = f \cdot k - 2o$

$w' = w/k$

$c' = c.$

Collect $k\ o\ m\ l$



$\mathrm{sig}(n, f, w, c) \rightarrow \mathrm{sig}'(n', f', w', c')$

The number of grouped streams to merge is given by k, the size of the groups in each grouped stream is given by m, the overlap between successive windows is given by o and the inter-window spacing is given by l.

$n' = n$

$f' = f \cdot k - 2lo/m$

$w' = w/k$

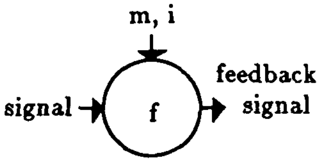
$c' = c.$

Temporal Sampling and Feedback Operators

The temporal sampling operators sample groups from each stream in the input signal according to the parameters. The output signal is composed of streams whose values are the sampled values from the input streams. These operators preserve width and change only the relative frequency. The temporal feedback operator denotes *state* in the program graph. Function state is maintained by changing the values on the feedback output stream to reflect the proper initialization and continuation of input values. This operator preserves all information in the signal and is purely denotational. The `Group` operator exploits temporal locality of reference; the `Sample` operator exploits temporal periodicity of reference.

Feedback $m\ i$

sig(n, f, w, c)→sig$'(n', f', w', c')$



This operator signifies that every mth token on each stream in the input signal is replaced with i in each output stream, where i is an object of size and dimension n.

$n' = n$

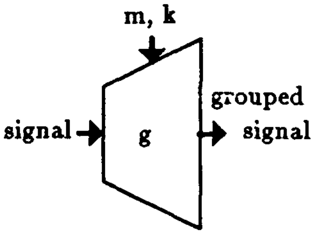
$f' = f$

$w' = w$

$c' = c.$

Group $m\ k$

sig(n, f, w, c)→sig$'(n', f', w', c')$



The offset of the group (temporal window) sampled from the beginning of the stream is given by m; the size of the window is given by k.

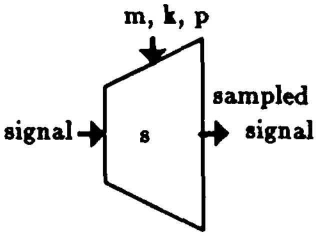
$n' = n$

$f' = 0$

$w' = w$

$c' = c.$

Sample $m\ k\ p$



sig$(n, f, w, c) \rightarrow$sig$'(n', f', w', c')$

The offset of each window sampled from the beginning of the stream is given by m, the size of the window is given by k and the inter-window spacing between successive windows is p.

$n' = n$

$f' = f \cdot km/l$

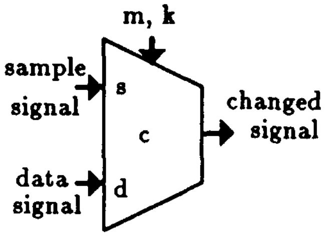
$w' = w$

$c' = c$.

Temporal Changing Operators

The temporal changing operators replace groups of values on each stream in the input data signal with groups from the input sample stream according to the parameters. The new streams with changed values form the output signal. These operators preserve width and frequency information. The Change operator exploits temporal locality of reference; the Overwrite operator exploits temporal periodicity of reference.

Change $m\ k$



sig$(n_d, f_d, w_d, c_d) \times$ sig(n_s, f_s, w_s, c_s)
$\rightarrow$sig$'(n', f', w', c')$

The selected groups of values (temporal windows) which are changed on the input data stream are given by m and k, where m is the offset of the window from the beginning of the stream and k is the size of the window.

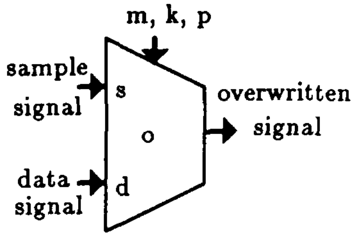
$n' = n_d$

$f' = f_d$

$w' = w_d$

$c' = c_d$.

Overwrite m k p



$$\text{sig}(n_d, f_d, w_d, c_d) \times \text{sig}(n_s, f_s, w_s, c_s)$$
$$\rightarrow \text{sig}'(n', f', w', c')$$

The selected groups of values which are changed on the input streams are given by m, k and p, where m is the offset from the beginning ^f the stream, k is the size of the window and p is the inter-window spacing.

$n' = n_d$

$f' = f_d$

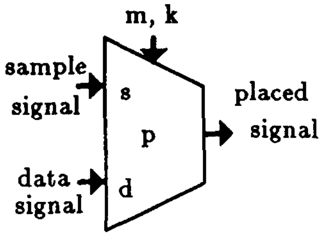
$w' = w_d$

$c' = c_d$.

Temporal Placing Operators

The temporal placing operators insert groups of values on each stream in the input data signal with groups from the input sample stream according to the parameters. The new streams with inserted values form the output signal. These operators preserve width and raise or preserve frequency. The `Place` operator exploits temporal locality of reference; the `Intersperse` operatoı exploits temporal periodicity of reference.

Place m k



$$\text{sig}(n_d, f_d, w_d, c_d) \times \text{sig}(n_s, f_s, w_s, c_s)$$
$$\rightarrow \text{sig}'(n', f', w', c')$$

The group of values (temporal window) inserted on the output streams is given by m and k, where m is the offset from the beginning of the stream and k is the size of the window.

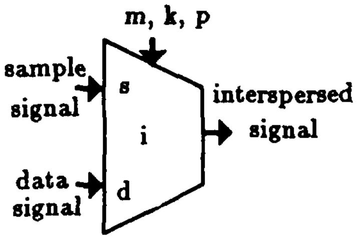
$n' = n_d$

$f' = f_d$

$w' = w_d$

$c' = c_d$.

Interspersе $m\ k\ p$



$\mathrm{sig}(n_d, f_d, w_d, c_d) \times \mathrm{sig}(n_s, f_s, w_s, c_s)$
$\rightarrow \mathrm{sig}'(n', f', w', c')$

The groups of values inserted on the output streams are given by m, k and p, where m is the offset from the beginning of the stream, k is the size of the window and p is the inter-window spacing.

$n' = n_d$

$f' = f_d + m$

$w' = w_d$

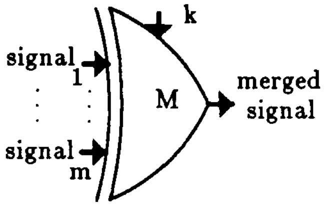
$c' = c_d$.

4.3.3 Channel Operators

The channel operators combine or distribute lines of processing from different, independent channels so that functions common to both channels can either be implemented in one task which processes both channels independently, or in separate tasks which process the channels in parallel. The number of combined streams is the number of tasks which process all streams. In the dataflow model, a tag is kept so that values will "remember" which stream they "belonged" to, so that independent processing can take place. This tag can be derived from the event number tag provided by the architectural model.

Combining different signals involves taking the union of the channel designators for the input signals in such a way that the correspondence between streams in the signal and channel information in the channel designator is easily recoverable. The actual number of channels designated in the output signal is the total number of channels designated by all input signals; the ordering of the output channel designator determines correspondence of streams to channels. When distributing signals from different lines of processing, the channel information is recovered by decomposing the channel designator. The relative frequency of the signal is changed, since streams are merged. Basic object information is unchanged.

Merge k



$\mathrm{sig}(n, f, w, c_1) \times \ldots \times \mathrm{sig}(n, f, w, c_m)$
$\rightarrow \mathrm{sig}'(n', f', w', c')$

k is the number of streams into which all input streams from all input signals are combined, *i.e.*, the size of the output signal.

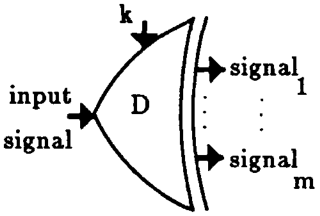
$n' = n$

$f' = f \cdot k$

$w' = w \cdot m/k$

$c' = \bigcup_{i=1}^{m} c_i$.

Distribute k



$\mathrm{sig}(n, f, w, c)$
$\rightarrow \mathrm{sig}'(n', f', w', c'_1) \times \ldots \times \mathrm{sig}'(n', f', w', c'_m)$

k is the number of streams which must be distributed into different output signals, *i.e.*, the size of the input signal.

$n' = n$

$f' = f/k$

$w' = w \cdot k/m$

$c'_i =$ (decompose $c\ i$).

Chapter 5

Transformation Rules

This chapter describes the graph constructor module of the compiler model described in Chapter 1. The graph constructor transforms a module of $\mathcal{PGL}$ code into the program graph representation composed of the graph objects described in Chapter 4. The graph constructor is an *interpreter* of $\mathcal{PGL}$ code, *i.e.,* there is a one-to-one correspondence between graph constructs and language constructs. The actual transformation is accomplished by a set of *Transformation Rules.* For each $\mathcal{PGL}$ construct, there is a rule which builds an associated graph construct and *wires* it into the graph for the module. These rules are formally described in the following sections.

5.1 Primary Program Transformation

The basic graph formation methodology is based on the dataflow nature of the applications, *i.e.,* functions precede others in the graph when the processing of the later functions depends on the outputs of the earlier functions. Therefore, arcs denote data dependencies between functions. The transformation of signal processing functions into nodes and arcs follows the structure of the code.

In order to re-use the same graph on differently-sized input signals, the signals on the arcs in the graph are not instantiated until simulation (evaluation) time. At simulation time, the dimension, size, frequency and channel information for all signals in the graph is derived by inference from the base input signals defined in the program. The data-routing operators are not dimension-independent, *i.e.*, the shape of the data is determined at compile time. The data-routing operators are size-independent, since instantiation of a signal is not performed at compile time.

The transformation of all constructs in the $\mathcal{PGL}$ program follows the data dependencies strictly. Data dependencies are determined in the environment determined by a block statement. Constants are the only global data in a program. The exact method of graph construction is left for a more detailed description. The following sections describe the transformation rules for the special $\mathcal{PGL}$ constructs which compile directly into graph constructs.

5.1.1 Transformation Rules for Iterators and Selectors

The following transformation rules define the graph constructor and establish the correspondence between $\mathcal{PGL}$ code and graph constructs. The transformation rules for selection constructs are straightforward. The transformation rules for iterations follow a three-step "enclosing" process in which

- the operators corresponding to all iterators, free variables in the body-statement and the composer are created (instantiation);
- the "k" parameters to all operators are wired together according to the mapping of input dimension to output dimension and free variable relationships;
- the graph for the body of the iteration is created.

In the following sections, examples of $\mathcal{PGL}$ code and the graph constructs which result from the transformation are presented. After each example, the general transformation rule which produced the transformation is defined.

Spatial Selection Operations

The following are spatial selection operations coded in $\mathcal{PGL}$:

```
def-select select-columns-2and3 = make-select ((0,0,0),(1,2,0));

def-select select-rows-4and8 = make-select ((3,1,3),(0,0,0));

def-replace replace-columns-3and4 = make-replace ((0,0,0),(2,2,0));

def-insert insert-after-row-3 = make-insert (1,3,2,0);

def-replace replace-rows-4and8 = make-replace ((3,1,3),(0,0,0));

def-insert insert-col-after-cols-4and8 = make-insert (2,4,1,4);
```

If we assume that **transform** is a function from an $m \times n$ matrix to another $m \times n$ matrix, then the following are examples of $\mathcal{PGL}$ code using the operators defined above.

```
def transform0-columns matrix =
  {columns-23 = select-columns-2and3 matrix;
   newcols = transform columns-23
   in
     replace-columns-3and4 newcols matrix};

def transform1-columns matrix =
  {columns-23 = select-columns-2and3 matrix;
   newcols = transform columns-23
   in
     insert-col-after-cols-4and8 newcols matrix};

def transform0-rows matrix =
  {rows-48 = select-rows-4and8 matrix;
   newrows = transform rows-48
   in
     insert-after-row-3 newrows matrix};

def transform1-rows matrix =
  {rows-48 = select-rows-4and8 matrix;
   newrows = transform rows-48
   in
     replace-rows-4and8 newrows matrix};
```

Assume in each case that the input is an 8×8 matrix. The first function selects the second and third columns, forming an 8×2 matrix. The function **transform** is applied to this matrix, yielding a new 8×2 matrix. The columns of this new matrix replace the third and fourth columns of the original matrix.

The second function inserts the columns of the new matrix after the fourth and eighth columns of the original matrix. The third function selects the fourth and eighth rows of the original matrix, inserting the rows of the new matrix after the third row of the original matrix. The last function selects the fourth and eighth rows of the original matrix, replacing these rows with the rows of the new matrix.

Figure 5.1 depicts the graphs resulting from these functions. The selectors which have non-zero values for any d_i in the window definition are transformed into Chooser operators; the others are transformed into Selector operators. The offset, window size and spacing parameters for the selectors are translated directly into the corresponding graph parameters.

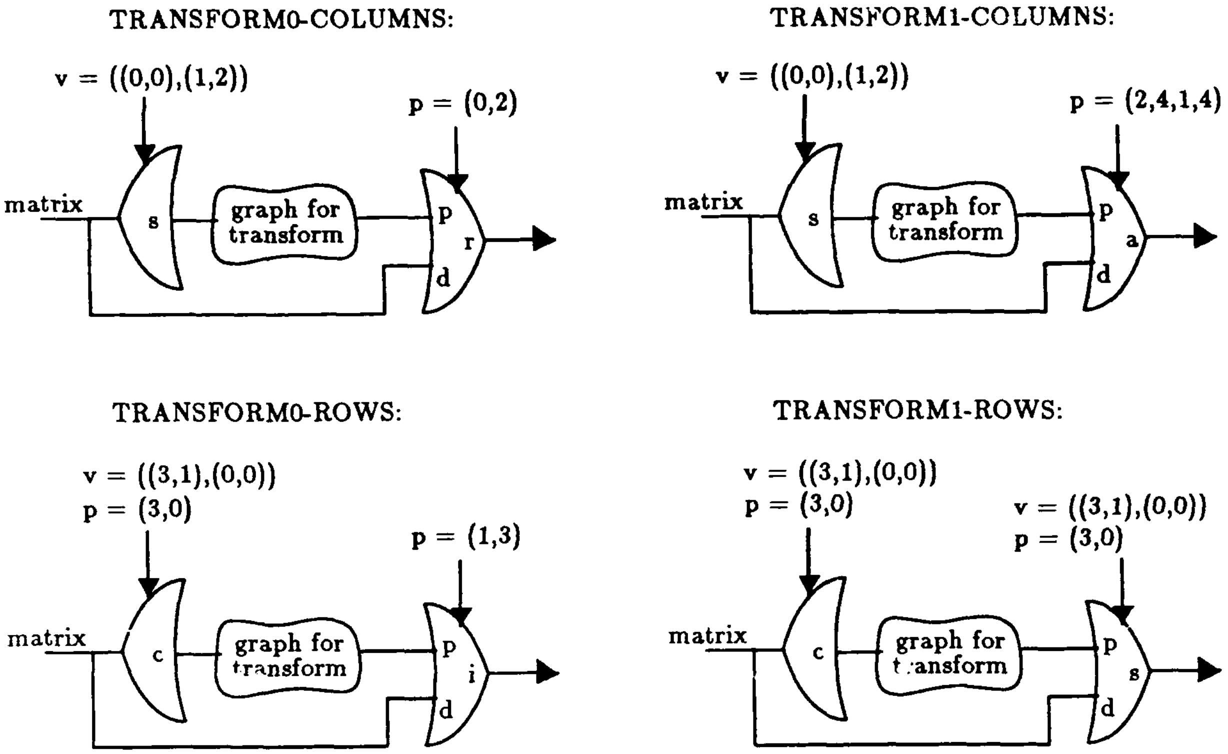


Figure 5.1: Program Graphs for Spatial Selections

The replacers and inserters which have non-zero values for any d_i in the window definition are transformed into `Substituter` and `Appender` operators, respectively; the others are transformed into `Replacer` and `Inserter` operators, respectively. The offset, window size, spacing parameters and dimension of insertion for the graph operators are derived directly from the corresponding language constructs. The following rules define the general transformation of spatial selection operations into graph constructs.

Transformation Rule 5.1 (Spatial Selection) *Given a $\mathcal{PGL}$ selector with window definition $w = (\ldots,(o_i,l_i,d_i),\ldots)$, a graph operator is instantiated according to one of the following cases:*

1. *If all $d_i = 0$, then a* `Selector` *is instantiated with parameter $v = (\ldots,(m_i,k_i),\ldots)$ where $m_i = o_i$ and $k_i = l_i$.*

2. *If any $d_i \neq 0$, then a* `Chooser` *is instantiated with parameters $v = (\ldots,(m_i,k_i),\ldots)$ and $p = (\ldots,p_i,\ldots)$ where $m_i = o_i$, $k_i = l_i$ and $p_i = d_i$.*

Transformation Rule 5.2 (Spatial Replacement) *Given a $\mathcal{PGL}$ replacer with window definition $w = (\ldots,(o_i,l_i,d_i),\ldots)$, a graph operator is instantiated according to one of the following* *cases:*

1. *If all* $d_i = 0$*, then a* `Replacer` *is instantiated with parameter* $p = (\ldots, p_i, \ldots)$ *where* $p_i = o_i$.

2. *If any* $d_i \neq 0$*, then a* `Substituter` *is instantiated with parameters* $v = (\ldots, (m_i, k_i), \ldots)$ *and* $p = (\ldots, p_i, \ldots)$ *where* $m_i = o_i$*,* $k_i = l_i$ *and* $p_i = d_i$.

Transformation Rule 5.3 (Spatial Insertion) *Given a* $\mathcal{PGL}$ *inserter with insertion mask* $w = (n, o, l, d)$*, a graph operator is instantiated according to one of the following cases:*

1. *If* $d = 0$*, then an* `Inserter` *is instantiated with parameters* $p = (r, s)$ *where* $r = n$ *and* $s = o$.

2. *If* $d \neq 0$*, then an* `Appender` *is instantiated with parameters* $p = (r, m, k, s)$ *where* $r = n$*,* $m = o$*,* $k = l$ *and* $s = d$.

Temporal Sampling Operations

The following are temporal sampling constructs coded in $\mathcal{PGL}$:

```
def-sample sample-4-offset-0 = make-sample (0,4,0);

def-sample sample-4-every-4 = make-sample (2,4,4);

def-change change-1-offset-4 = make-change (3,1,0);

def-change change-1-every-4 = make-change (2,1,4);

def-place place-1-offset-4 = make-place (4,1,0);

def-place place-1-every-4 = make-place (2,1,4);
```

If we assume that `filter` is a function which outputs one token for every four input, so that the relative frequency of the output stream is one fourth of that of the input stream, then the following are examples of $\mathcal{PGL}$ code using the operators defined above:

```
def transform0-stream in-stream =
  {stream-of-4 = sample-4-offset-0 in-stream;
   new-stream = filter stream-of-4
   in
     change-1-offset-4 new-stream in-stream}
```

```
def transform1-stream in-stream =
  {stream-of-4 = sample-4-offset-0 in-stream;
   new-stream = filter stream-of-4
   in
     place-1-offset-4 new-stream in-stream}

def transform2-stream in-stream =
  {stream-4-every-4 = sample-4-every-4 in-stream;
   new-stream = filter stream-4-every-4
   in
     change-1-every-4 new-stream in-stream};

def transform3-stream in-stream =
  {stream-4-every-4 = sample-4-every-4 in-stream;
   new-stream = filter stream-4-every-4;
   in
     place-1-every-4 new-stream in-stream}
```

The first function creates a finite stream of size four by sampling from the input stream then applies `filter` to that stream yielding a finite stream of size one. The output value on this stream replaces the fourth value in the original input stream. The second function inserts this output value after the fourth value in the original input stream.

The third function creates an infinite stream by sampling four out of every four values from the input stream and applies `filter` to that stream yielding an infinite stream. The output values on this stream replace every fourth value in the original input stream starting with the third. The fourth function inserts the output values after every fourth value starting with the third.

Figure 5.2 depicts the graphs which result from these functions. The samplers which have non-zero *d* values in the window definition are transformed into `Sampler` operators; the others are transformed into `Grouper` operators. The offset, window size and spacing parameters for the samplers are translated directly into corresponding graph parameters.

The changers and placers which have non-zero *d* values are transformed into `Overwriter` and `Intersperser` operators, respectively; the others are transformed into `Changer` and `Placer` operators, respectively. The offset, window size and spacing parameters for the graph operators are derived directly from the language constructs. The following rules define the general transformation of temporal sampling operations into graph constructs.

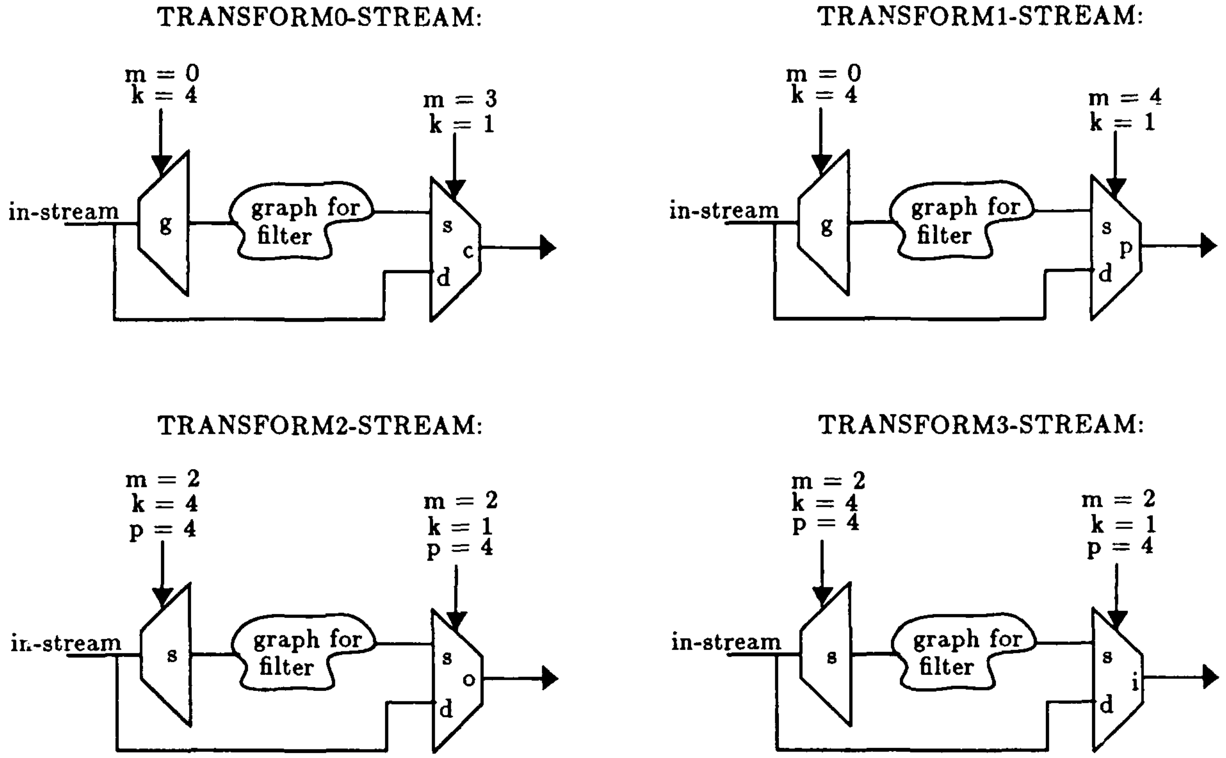


Figure 5.2: Program Graphs for Temporal Samplings

Transformation Rule 5.4 (Temporal Sampling) *Given a $\mathcal{PGL}$ sampler with window definition $w = (o, l, d)$, a graph operator is instantiated according to one of the following cases:*

1. *If $d = 0$, then a* `Grouper` *is instantiated with parameters (m, k) where $m = o$ and $k = l$.*

2. *If $d \neq 0$, then a* `Sampler` *is instantiated with parameter*^{c} $'$ $k, p)$ *where $m = o$, $k = l$ and $p = d$.*

Transformation Rule 5.5 (Temporal Changing) *Given a $\mathcal{PGL}$ changer with window definition $w = (o, l, d)$, a graph operator is instantiated according to one of the following cases:*

1. *If $d = 0$, then a* `Changer` *is instantiated with parameters m, k where $m = o$ and $k = l$.*

2. *If $d \neq 0$, then an* `Overwriter` *is instantiated with parameters (m, k, p) where $m = o$, $k = l$ and $p = d$.*

Transformation Rule 5.6 (Temporal Placing) *Given a $\mathcal{PGL}$ placer with window definition $w = (o, l, d)$, a graph operator is instantiated according to one of the following cases:*

1. *If $d = 0$, then a* `Placer` *is instantiated with parameters (m, k) where $m = o$ and $k = l$.*

2. *If $d \neq 0$, then an* Intersperser *is instantiated with parameters (m, k, p) where $m = o$, $k = l$ and $p = d$.*

Spatial Iteration

Two examples of spatial iterations coded in $\mathcal{PGL}$ are shown below:

```
def transform-0 matrix =
  {generate wbox from (gen-matrix-windows matrix) in
     w-transform0 wbox;
   compose comp-matrix-windows};

def-iter gen-matrix-windows =
  {window = ((2,0),(2,0));
   point-list = ((0,1),(0,0));
   order = (2,1);
   in
     make-iter window point-list order};

def-comp comp-matrix-windows =
  {window = ((1,3),(1,3));
   point-list = ((0,0),(0,0));
   order = (1,2);
   in
     make-comp window point-list order};

def transform-1 matrix1 matrix2 free-mat =
  {generate wdiv1,wdiv2 from (gen-matrix-divisions matrix1),
                             (gen-matrix-divisions matrix2) in
     w-transform1 wdiv1 wdiv2 free-mat;
   compose comp-matrix-divisions};

def-iter gen-matrix-divisions =
  {window = ((1,3),(2,0));
   point-list = ((0,0),(0,0));
   order = (2,1);
   in
     make-iter window point-list order};

def-comp comp-matrix-divisions =
  {window = ((2,0),(2,0));
   point-list = ((0,1),(0,0));
   order = (2,1);
   in
     make-comp window point-list order};
```

For the above examples, we assume that **w-transform0** is a function from a 2×2 matrix to another 2×2 matrix and **w-transform1** is a function from three 2×2 matrices to one 2×2 matrix. In both cases, the input matrix is an 8×8 matrix. The first function, **transform-0**, then iterates over 2×2 windows from the original matrix. These windows are chosen at every point along the column dimension and every other point along the row dimension. Therefore, there will be overlap on the second dimension and none on the first. The second dimension is traversed first to yield the list of matrices.

The list of result matrices is combined into a 16×8 matrix by distributing the elements of each result matrix over the specified positions in the output matrix, traversing the first dimension first. If the input matrix is

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix}$$

then the list of regions generated by the iterator is

$$\left(\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \begin{bmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{bmatrix}, \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix}, \ldots \right)$$

If the list of result objects is

$$\left(\begin{bmatrix} a1_{11} & a1_{12} \\ a1_{21} & a1_{22} \end{bmatrix}, \begin{bmatrix} a2_{11} & a2_{12} \\ a2_{21} & a2_{22} \end{bmatrix}, \begin{bmatrix} a3_{11} & a3_{12} \\ a3_{21} & a3_{22} \end{bmatrix}, \ldots \right)$$

then the output matrix is

$$\begin{bmatrix} a1_{11} & a9_{11} & a17_{11} & a25_{11} & a1_{12} & a9_{12} & a17_{12} & a25_{12} \\ a2_{11} & a10_{11} & a18_{11} & a26_{11} & a2_{12} & a10_{12} & a18_{12} & a26_{12} \\ & & & \vdots & & & & \\ a1_{21} & a9_{21} & a17_{21} & a25_{21} & a1_{22} & a9_{22} & a17_{22} & a25_{22} \\ a2_{21} & a10_{21} & a18_{21} & a26_{21} & a2_{22} & a10_{22} & a18 & a26_{22} \\ & & & \vdots & & & & \end{bmatrix}$$

The second function generates windows by selecting two on the column dimension and one on the row dimension with three element spacing at every point. Therefore, there is overlap on the second dimension and none on the first. The first dimension is traversed first in generating the list. The result matrices are combined into an 8 × 8 matrix. If an input matrix is

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix}$$

then the list of regions generated by the iterator is

$$\left(\begin{bmatrix} a_{11} & a_{12} \\ a_{51} & a_{52} \end{bmatrix}, \begin{bmatrix} a_{12} & a_{13} \\ a_{52} & a_{53} \end{bmatrix}, \begin{bmatrix} a_{13} & a_{14} \\ a_{53} & a_{54} \end{bmatrix}, \ldots \right)$$

If the list of result objects contains 2 × 2 matrices, these matrices are combined at every point along the column dimension and every other point along the row dimension. Therefore, there will be overlap on the second dimension and none on the first. Overlap of output matrices is assumed to cause an averaging of the overlapped elements.

Figure 5.3 shows the program graphs which result from the transformation of these functions. The iterators which have non-zero d values in the window definition are transformed into `Divider` operators; the others are transformed into `Splitter` operators. The composers which have non-zero d values in the window definition are transformed into `Recomposer` operators; the others are transformed into `Combiner` operators. Free variables in the body statement are transformed into `Duplicator` operators.

Because the k parameters of the operator pairs are connected, the transformation rule for spatial iterations transforms the iteration construct as a unit. The following rule defines the general transformation of spatial iteration constructs into graph constructs.

Transformation Rule 5.7 (Spatial Iteration) *Given a $\mathcal{PGL}$ spatial iteration composed of: the iterators* $((w_1, p_1, o_1) \ldots (w_n, p_n, o_n))$ *where* $w_i = (\ldots, (l_{ij}, d_{ij}), \ldots)$, $p_i = (\ldots, (o_{ij}, s_{ij}), \ldots)$,

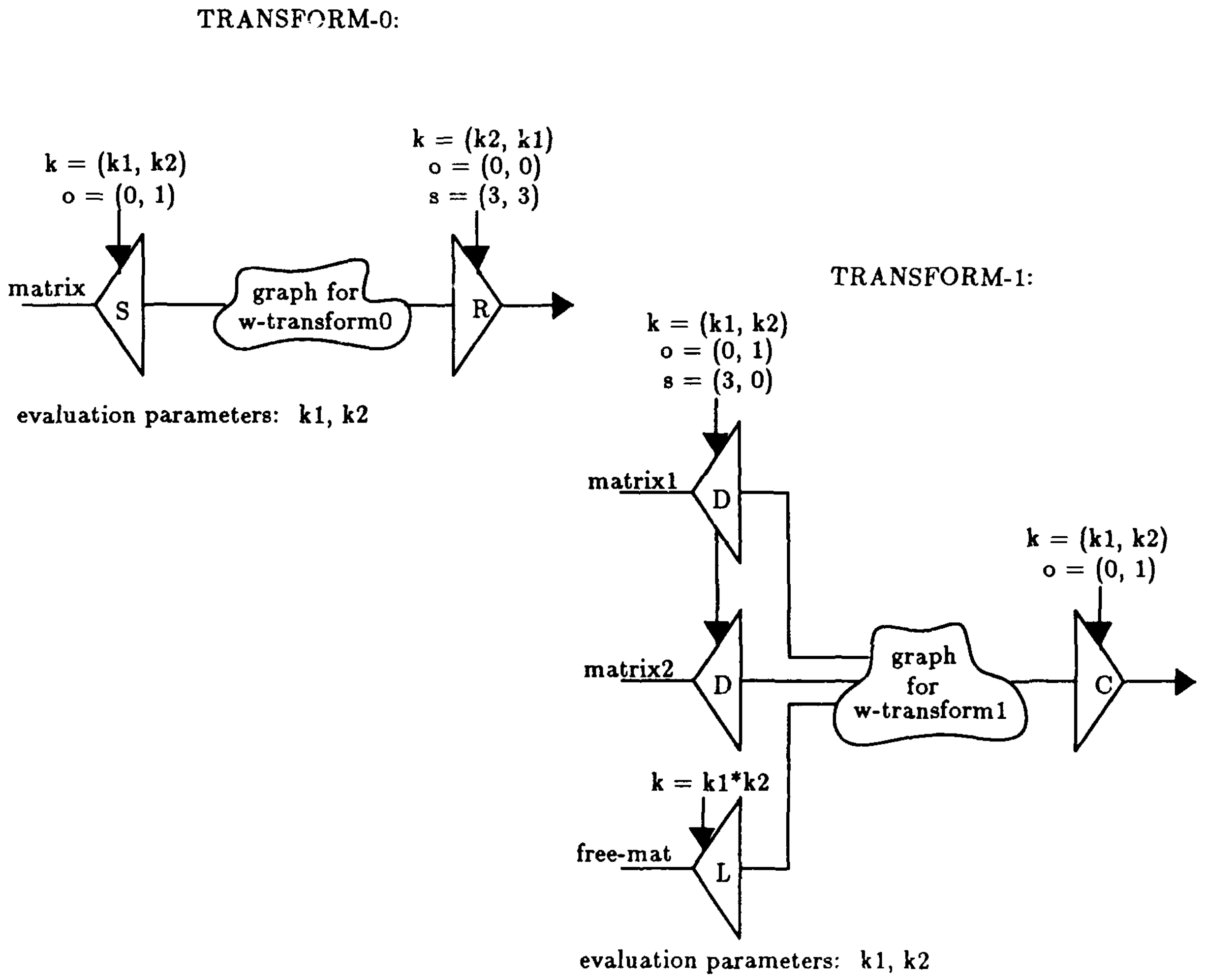


Figure 5.3: Program Graphs for Spatial Iterations

and $o_i = (\ldots, n_{ij}, \ldots)$*; and the composer* (w_c, p_c, o_c) *where* $w_c = (\ldots, (l_{cj}, d_{cj}), \ldots)$, $p_i = (\ldots, (o_{cj}, s_{cj}), \ldots)$, *and* $o_i = (\ldots, n_{cj}, \ldots)$, *the three step rule proceeds as follows:*

1. *Instantiation of operators proceeds in three steps:*

   (a) Instantiation of iterators. *For each iterator i:*

      i. *If all* $d_{ij} = 0$, *then a* `Splitter` *with parameters* o_{Si} *and* k_{Si} *is created where* $o_{Sij} = l_{ij} - s_{ij} - 1$ *for dimension* j *of the input. If any* $l_{ij} = 0$, *then* $o_{Sij} = N$ *where* N *is a special constant in the graph denoting that the size of the window is the size of the matrix on dimension* j.

      ii. *If any* $d_{ij} \neq 0$, *then a* `Divider` *with parameters* o_{Si}, k_{Si} *and* s_{Si} *is created where* $o_{Sij} = l_{ij} - s_{ij} - 1$ *and* $s_{Sij} = d_{ij}$ *for dimension* j *of the input. If any* $l_{ij} = 0$, *then* $o_{Sij} = N$ *where* N *is a special constant in the graph denoting that the size of the window is the size of the matrix on dimension* j.

   (b) Instantiation of the composer. *For composer c:*

      i. *If all* $d_{cj} = 0$, *then a* `Combiner` *with parameters* o_C *and* k_C *is created where* $o_{Cj} = l_{cj} - s_{cj} - 1$ *for dimension* j *of the output. If any* $l_{cj} = 0$, *then* $o_{Cj} = N$ *where* N *is a special constant in the graph denoting that the size of the window is the size of the matrix on dimension* j.

      ii. *If any* $d_{cj} \neq 0$, *then a* `Recomposer` *with parameters* k_C, o_C *and* s_C *is created where* $o_{Cj} = l_{cj} - s_{cj} - 1$ *and* $s_{Cj} = d_{cj}$ *for dimension* j *of the output. If any* $l_{cj} = 0$, *then* $o_{Cj} = N$ *where* N *is a special constant in the graph denoting that the size of the window is the size of the matrix on dimension* j.

   (c) Instantiation of duplicators. *For each free variable in the body statement of the iteration, a* `Duplicator` *is created with parameter* k_D.

2. *Wiring of k parameters is performed in three steps:*

   (a) Wiring of enclosing operators. *For each iterator* i, $k_{Cn_{cj}}$ *is wired to* $k_{Sin_{ij}}$ *for dimension* j *of the output. If* n_{ij} *does not exist, then* $k_{Cn_{cj}}$ *is wired to a null value.*

   (b) Wiring of several iterators. *For each pair of iterators* (i, i'), $k_{Sn_{ij}}$ *is wired to* $k_{Sn_{i'j}}$. *If* $n_{i'j}$ *does not exist, then* $k_{Sn_{ij}}$ *is wired to a null value.*

   (c) Wiring of duplicators. *For each duplicator,* k_D *is wired to the symbolic product of all the* k_{Sij} *for any iterator* i *and all dimensions* j *for which* k_{Sij} *is non-null.*

3. *The graph for the iteration body is created, with its inputs connected to the outputs of the iterating or duplicating operators and with its output connected to the input of the composing operator.*

Temporal Iteration

Two examples of temporal iterations coded in $\mathcal{PGL}$ are shown below:

```
def accum-0 in-stream =
  {init = 0
   in
     {repeat slice over sample-4-every-4 in-stream in
        tfold sum-elements0 slice init;
      collect compose-every-4}};

def-repeat sample-4-every-4 =
  {window = (4,0);
   point-list = (0,4);
   in
     make-repeat window point-list};

def-collect compose-every-4 =
  {window = (1,0);
   point-list = (0,1);
   in
     make-collect window point-list};

def accum-1 in-stream free-val =
  {init = 0
   in
     {repeat slice over group-4-every-2 in-stream in
        tfold sum-elements1 slice init free-val;
      collect ungroup-every-4}};

def-repeat group-4-every-2 =
  {window = (4,4);
   point-list = (1,2);
   in
     make-repeat window point-list};

def-collect ungroup-every-4 =
  {window = (1,4);
   point-list = (0,1);
   in
     make-collect window point-list};
```

In these examples, we assume that `in-stream` and `free-val` are streams of real numbers, `sum-elements0` has signature $r \times r \rightarrow r$ and `sum-elements1` has signature $r \times r \times r \rightarrow r$. The first function, `accum-0`, performs a temporal fold over groups of four elements selected from the input stream every four elements so that there is no overlap of temporal windows. These four elements are summed by `sum-elements` with initial value zero to yield a result token. This token is output to the output stream and the sum is re-initialized for the next folding of `sum-elements`.

The second function, `accum-1`, performs a temporal fold over groups of four elements selected form the input stream every two elements with inter-window spacing equal to four so that there is an overlap of two tokens from the input stream. The groups of four tokens are summed by `sum-elements` with initial value zero for the first invocation at each defined point in the stream. The sum is output to the output stream but is not re-initialized for the subsequent invocations. For each group of four input tokens, one result token is output.

The relative frequency for each function will therefore be one-fourth of that of the input stream for the first function and one-half of that of the input stream for the second function (because of the overlap of windows). Figure 5.4 shows the program graphs which result from *the transformation* of these functions. The temporal iterators which have non-zero d values in the window definition are transformed into `Rotator` operators; the others are transformed into `Disperser` operators. The temporal composers which have non-zero d values in the window definition are transformed into `Collector` operators; the others are transformed into `Releaser` operators. Free variables inside the body statement are transformed into `Repeater` operators.

Because the k parameters of the operator pairs are connected, the transformation rule for temporal iterations transforms the iteration construct as a unit. The following rule defines the general transformation of temporal iteration constructs into graph constructs.

Transformation Rule 5.8 (Temporal Iteration) *Given a temporal iteration composed of: iterators* $(w_1, p_1) \ldots (w_n, p_n)$ *where* $w_i = (l_i, d_i)$ *and* $p_i = (o_i, s_i)$*; and the composer* (w_c, p_c) *where* $w_c = (l_c, d_c)$ *and* $p_c = (o_c, s_c)$*, the three step rule proceeds as follows:*

1. *Instantiation of operators proceeds in three steps:*

   (a) Instantiation of iterators. *For each iterator i:*

   i. *If* $d_i = 0$*, then a* `Disperser` *with parameters* k_{Di}, m_{Di} *and* o_{Di} *is created where* $o_{Di} = l_i - s_i$ *and* $m_{Di} = l_i$.

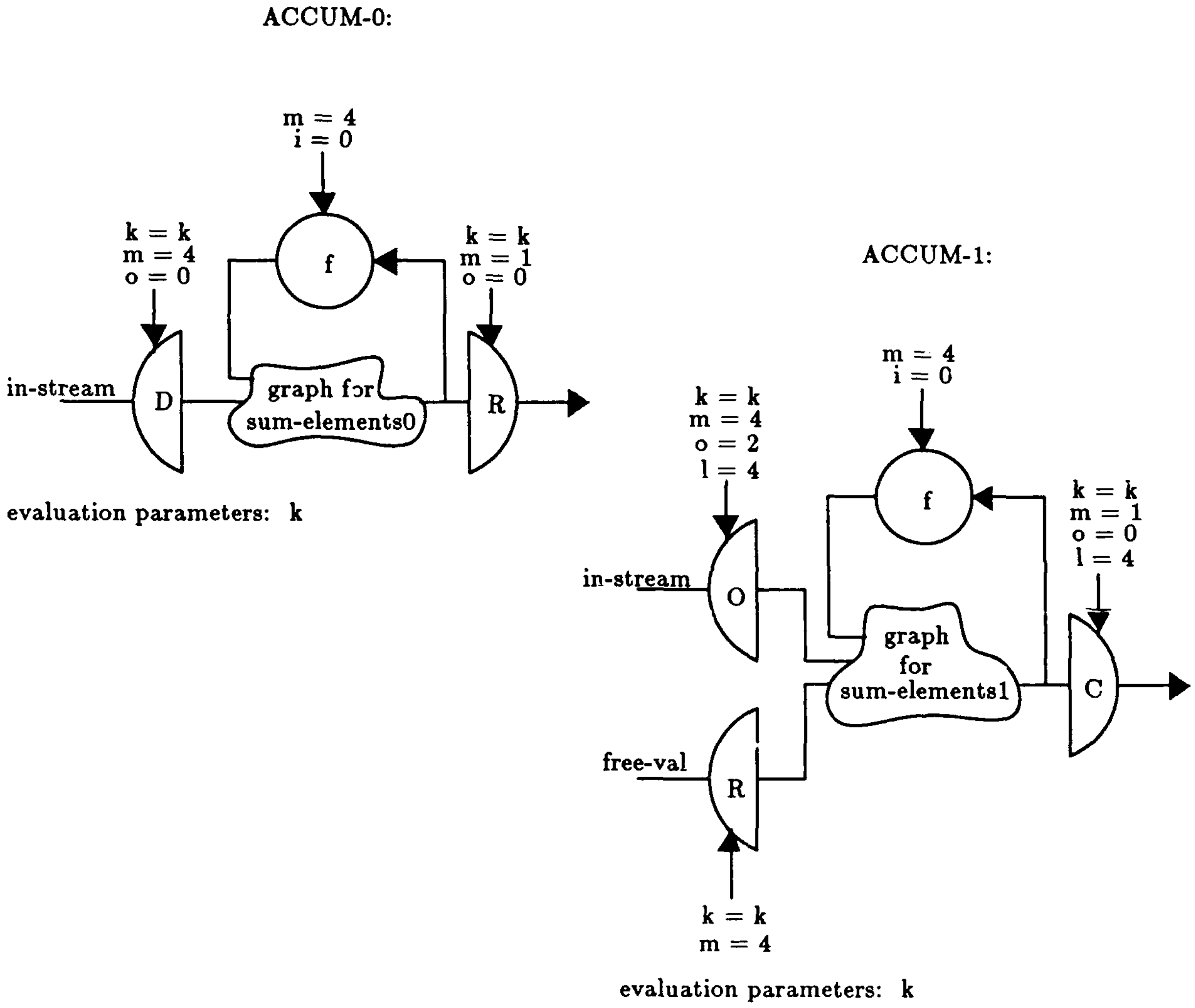


Figure 5.4: Program Graphs for Temporal Iterations

   ii. *If $d_i \neq 0$, then a* `Rotator` *with parameters k_{Di}, o_{Di}, m_{Di}, and l_{Di} is created where $o_{Di} = l_i - s_i$, $m_{Di} = l_i$ and $l_{Di} = d_i$.*

   (b) Instantiation of the composer. *For composer c:*

   i. *If $d_c = 0$, then a* `Releaser` *with parameters k_R, m_R, and o_R is created where $o_R = l_c - s_c$ and $m_R = l_c$.*

   ii. *If $d_c \neq 0$, then a* `Collector` *with parameters k_R, o_R, m_R and l_R is created where $o_R = l_c - s_c$, $m_R = l_c$ and $l_R = d_c$.*

   (c) Instantiation of repeaters. *For each free variable in the body statement of the iteration, a* `Repeater` *is created with parameters k_P and m_P where $m_P = l_c$.*

2. *Wiring of k parameters is performed in three steps:*

   (a) Wiring of enclosing operators. *For each iterator i, k_{Di} is wired to k_R.*

   (b) Wiring of several iterators. *For each pair of iterators (i, i'), k_{Di} is wired to $k_{Di'}$.*

   (c) Wiring of repeators. *For each repeater, k_P is wired to k_{Di} for any iterator i.*

3. *The graph for the* `tfold` *statement is constructed as follows:*

   (a) *The graph for the fold-function is created with stream inputs connected to the outputs of the iterating and repeating operators and with its output connected to the input of the composing operator.*

   (b) *A* `Feedback` *operator is created with parameters m_f and i_f where $m_f = l_i$ for any iterator i and i_f is the dimension and size list of the value of* `init` *supplied in the code. If this information cannot be determined at compile time then the value of i_f is not supplied until evaluation time. The output of the fold-function graph is connected to the input of the* `feedback` *operator. The output of the* `feedback` *operator is connected to the input of the fold-function designated by the* `init` *keyword in the $\mathcal{PGL}$ code.*

5.2 Secondary Transformation Rules

The secondary transformation rules implement the first pass optimization module of the compiler described in the introduction. At this point, the entire graph of the application is constructed. In this phase, `Repeat`, `Distribute` and `Merge` operators are inserted into the graph.

The **Repeat** operator is used when two inputs to a function have different relative frequencies, and an adjustment to one of them must be made. The **Distribute** and **Merge** operators are used when common portions of the graph are executed on data of the same size and therefore can be merged into one task. A sample portion of a graph after insertion of these operators is included in Appendix B. Full specification of this portion of the compiler is left for future work.

Chapter 6

Conclusion

The program graph representation and the language $\mathcal{PGL}$ described in the preceding chapters are designed to support the automatic or algorithmic partitioning of the computations in an application into tasks to run on a multiprocessor. Simplification of the partitioning process is desirable for many reasons; chief among them are the complexity of the partitioning task and the difficulty in finding a partition which is optimal or near-optimal. For most machines and applications today, the burden of partitioning is placed mostly on the programmer and is accomplished by means of *ad hoc* techniques useful only in specific cases and without proof of optimality.

Several questions can be asked about the automatic decomposition of an application program into tasks; these are listed below:

- At what point in the compile-execute cycle should decomposition take place? Should decomposition be performed *statically*, *i.e.*, at compile time, or *dynamically*, *i.e.*, at run time?

- How thoroughly should the program be partitioned, *i.e.*, should a program be decomposed in the easiest manner possible, or should a better or possibly optimal partition be attempted?

- What are the criteria for an optimal partition, *i.e.*, what program characteristics or performance measures determine the requirements for optimality? For example, should optimization consist of minimizing latency, maximizing throughput, minimizing the number of tasks, minimizing the amount of memory used for the queueing buffers at task inputs, or a combination of all four?

- To what extent, if any, must machine characteristics be known at the time of decomposition? If decomposition is dynamic, there must be some form of resource management; if static, then how much resource information should be included in finding a good partition? Or should task structure be strictly a function of computational structure? If machine characteristics are taken into account, for how general a class of machines can we design a partitioning algorithm?

- How general should a partitioning algorithm be, *i.e.*, how general a class of applications should be covered by the algorithm? Can decomposition be performed on applications with hard constraints, for example, real-time constraints, or is any partitioning algorithm which could be developed limited to a very simple class of applications with very simple structure and small processing requirements?

- What level of granularity is assumed? Is it possible to find an algorithm to partition a program at variable levels of granularity, or is any scheme for automating the decomposition process limited to a particular task size or unit of processing?

The focus of current research in these areas leans toward new programming paradigms, as well as new tools for support of programming for parallel machines. The development of new programming paradigms leads to new languages and representations. The point of view taken in this work is that these new languages and representations can instrinsically and sufficiently answer the questions posed above. In this way, automatic partitioning can be achieved more easily without burdening the programmer. The language $\mathcal{PGL}$ is designed with a set of answers to the questions posed above; these are outlined in the next few paragraphs.

Partitioning of an application coded in $\mathcal{PGL}$ is not dynamic, however, it is also not completely static in the sense that it is compiled into a representation which only partially determines task structure. In this way, decomposition of a $\mathcal{PGL}$ program is performed at *simulation* time, where simulation time is an intermediate stage in which a given task structure is evaluated before the final partition is chosen.

A main motivation for the scheme described in this work is that an optimal or near-optimal partition is desired. Criteria for the optimal partition is partially known in the sense that it is based on certain aspects of the task structure which are quantifiable given the graph representation, however the actual optimization function is not totally determined and is suppled by the programmer or system designer. Machine characteristics are not a large factor in evaluating a program partition beyond the general characteristics exhibited by the class of machines described in the introduction.

The focus of this work has been on signal processing applications so that the specific task of designing a language was more easily achievable. In these applications, real-time constraints are of a specific type and known at compile time, so that a framework for expressing them was more easily developed. The real-time constraints in these applications involved relative frequencies of inputs, all of which are derivable from the base input streams. Also, the patterns of data and stream access in the signal processing applications simplified the development of a basic set of operators. This set of operators is also comprehensive enough to be useful. $\mathcal{PGL}$ and its graph representation are therefore application-specific tools which are designed to illustrate a more general methodology.

This work has also focused on coarse-grain parallelism due to constraints imposed by the target architecture. To a certain extent, granularity is variable, since the programmer can arbitrarily designate base functions in the graph. By leaving this option open to the programmer, coarser grains are included in the programming paradigm itself. Programming for finer grains in $\mathcal{PGL}$ does not involve further designation, however, below a certain grain size, the size of the program may become unwieldy. The paradigm is therefore not exclusive of fine-grain parallelism, however, such programming is more difficult to achieve and the graph representation is less efficient than a traditional dataflow graph.

The goal of this thesis was to show that a language and a graph representation with the above properties are both feasible and desirable. The desirability of the tool is presented in the introduction and design motivations are presented in the second chapter. The feasibility of the language $\mathcal{PGL}$ is shown by the transformation and composition rules presented in Chapter 5. The correctness of these rules is not proven, however, the demonstration of the use of language and graph is provided in Appendices B and C using parts of the space-based radar application as examples. This demonstration shows both feasibility and desirability in at least one specific case.

6.1 Future Work

6.1.1 Implementation

Further work must be done on proving correctness of the system, either through actual implementation or by theoretical proof. The latter is not practical, since the system was designed to be implemented and not to be theoretically understood. Since the system was designed to be implemented, the first step is the implementation of the $\mathcal{PGL}$ compiler front-end, *i.e.*, the graph constructor. This is a literal implementation of the transformation rules described in Chapter 5, which for reasons related to timing of the work, was not completed before the writing of the thesis.

6.1.2 Decomposition

The next step in the implementation process is the implementation of the latter phases and modules of the compiler described in the introduction. This is not the concern of this thesis; however, basic ideas for each module were considered and are presented below:

- *Linker.* The Linker module merges separate program graphs together into a final graph representing the entire application. This task is straightforward and presents no major difficulties.

- *First-pass optimization.* At this phase of the compiler, peephole optimizations on the structure of the graph as a whole can be applied. These may involve merging sequential operaters into one operator or possibly eliminating redundant operators in the graph.

- *Decomposition.* This is the most complicated phase of the compiler and has several unsolved issues. The first is the definition of optimality and the method of evaluating a given partition. In terms of the program graph, this means for a given set of operator parameter values, a well-defined algorithm must be developed which finds the "goodness" of the task structure corresponding to those values.

  Another unsolved issue is the design of the user interface to the tool. An interactive decomposer is envisioned; however, for large applications such as the radar application, the number of operator parameter values which the programmer must supply becomes too large for convenience. An alternative is the development of iterative algorithms which evaluate the partition and then supply new parameter values based on the evaluation.

Development of these algorithms is also a future possibility; techniques such as simulated annealing may prove useful in this area.

Program decomposition was considered in this work only to the extent necessary to determine what information must be included in the program graph to make algorithmic partitioning possible. Further work in this area could involve more complete demonstrations that all information necessary to find an optimal or near-optimal partition of an application program is contained in the program graph.

6.1.3 Improvements

Several improvements can be made to the currently defined system. The first of these is expansion of the applicability of the system to reach higher levels of generality. Other applications can be tested against the current capability of the system. A more complete set of operators could be developed to meet the needs of more complex applications. Higher level data abstractions are also not included in the current language and graph design; this is an extremely important area which could greatly improve capability of the system.

Since the program graph only expresses task structure, the program graph could also be used for automatic code generation. This assumes that operators and functions are well-defined enough that translation is possible.

6.2 Conclusion

This thesis has described an attempt to design a language and graph representation to be useful in the partitioning of large signal processing application programs into tasks to run on a multiprocessor. As parallel processing continues to widen in applicability and appeal, the demand for tools to support programming will also increase. It is hoped that the work described here will contribute to the growing body of research in this area.

Appendix A

Application Description

A.1 Space Based Radar

The application presented here is based on the processing needs of a typical surveillance SBR system with a displaced phase center antenna (DPCA) [5, 17]. In this appendix, the signal processing functions are presented. A block diagram of the DCPA SBR system is shown in Figure A.1.

A.1.1 The DPCA SBR system

In a displaced phase center antenna system, satellite motion, *i.e.*, the motion of the SBR platform, consists of two components, one parallel to the antenna aperture and the other perpendicular to the antenna aperture. These components have the undesired effects of spreading the spectrum of the clutter returns (less clear doppler) and shifting the mean of the spectrum. DCPA compensates for the component of the satellite velocity parallel to the antenna aperture by displacing the phase center of the antenna in the direction opposite to the velocity component, so that the antenna appears stationary to the clutter. To eliminate the stationary clutter, a simple moving target indicator (MTI) can be used. The application described here uses a two-pulse canceller.

A.1.2 Waveforms

The baseline system uses a pulse doppler waveform, meaning that discrete time sampling of the echo signal is performed. Sampling is done on transmission, *i.e.*, pulses are transmitted instead of a continuous wave. This is done to obtain target range as well as doppler frequency. Pulse

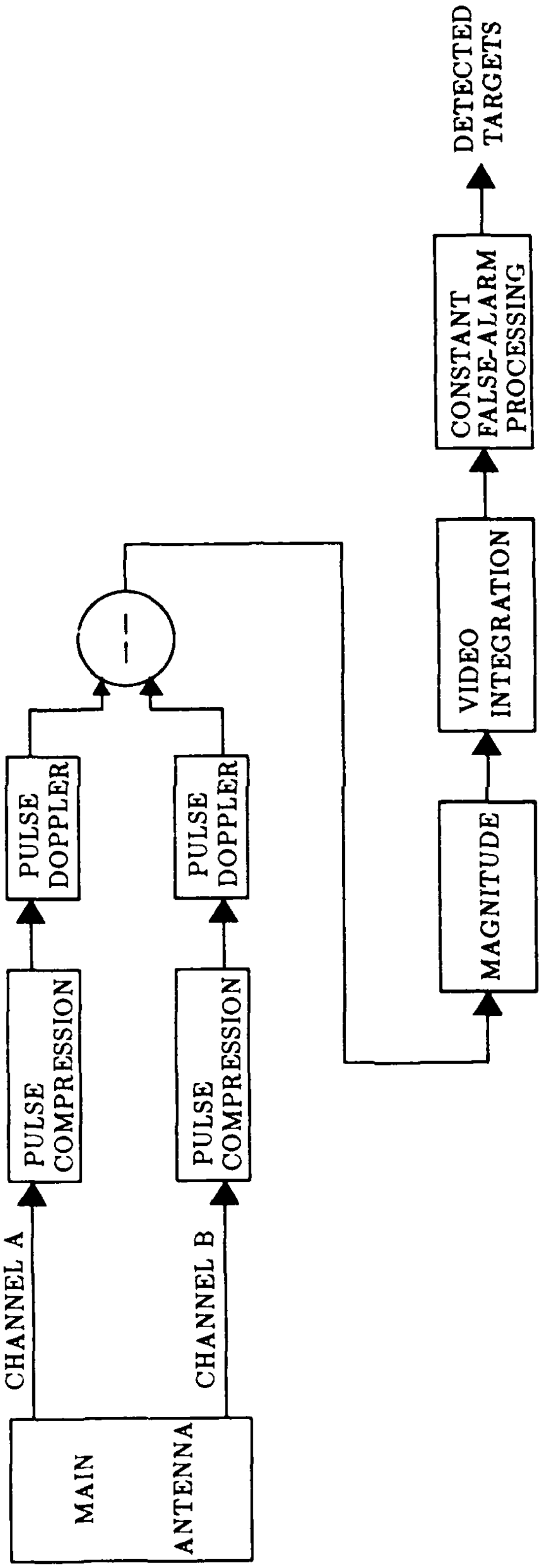


Figure A.1: DCPA System Block Diagram

compression is used to cope with the peak power constraints of a solid-state transmitter.

A typical pulse doppler air surveillance waveform is shown in Figure A.2. A chirp (linear FM) waveform is used. The basic waveform is shown in Figure A.2a. The transmitter is on for T_t seconds. At the end of this time, the transmitter is turned off and the receiver is turned on for T_r seconds. During this time, N_{rc} pulses are received. Each pulse is sampled N_r times. The basic waveform consists of M_T bursts, which are incoherently integrated (envelope detected and summed). Frequency diversity is used to reduce the fluctuation loss. Constant false-alarm (CFAR) processing is performed after video integration in each doppler channel.

In the waveform shown in Figure A.2b, groups of M_c bursts of the basic waveform are coherently integrated from burst to burst. In general, the total number of bursts in a dwell is given by $M_T = M \cdot M_c$, where M is the number of different RF carrier frequences used and M_c is the number of bursts which are coherent (at the same frequency). The total dwell time T_d is therefore equal to $M_T(T_r + T_t)$.

A.1.3 Signal Processing Functions

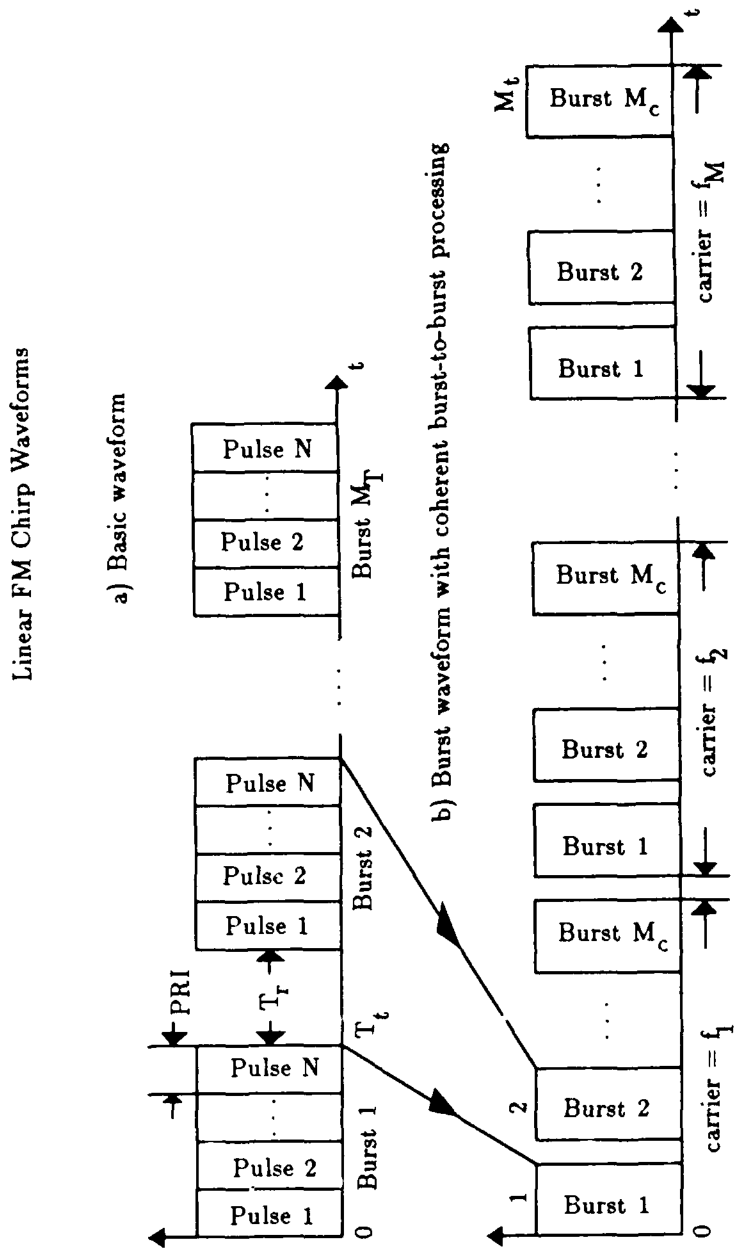
The rate $(1/T_d)$ and size of the data input from the sensors determine the throughput and memory requirements of the signal processing functions. The rate and size of the input data is determined by $1/T_d$ and the number of range and doppler cells. The number of range cells is given by:

$$N_r = \frac{RangeWindow}{RangeResolution} = \frac{PRI}{pulsewidth} = PRI \cdot B_{chirp}$$

where PRI is the pulse repetition interval and B_{chirp} is the linear chirp bandwidth. The maximum potential range window is equal to the PRI minus the pulsewidth. Since the pulsewidth is usually much smaller than the PRI, the range window is approximately equal to the PRI.

The number of doppler filters (FFT size) is determined by the number of received coherent time samples. The number of received samples from a single burst is given by $N_{rc} = T_r \cdot PRF$. When the burst waveform with coherent burst-to-burst processing is used, the number of doppler filters is given by $N_D = N_{rc} \cdot M_c$. In order to take advantage of the entire coherent time which is available, a long burst is used with a waveform such that the system can sample with the PRI. Sampling thus starts after time T_t, *i.e.*, the time that the first pulse arrives back from the target.

Data will therefore be received from the sensors after each burst in the following pattern: every T_t seconds, the receiving process will format N_{rc} samples at a rate of one sample every



Linear FM Chirp Waveforms

a) Basic waveform

b) Burst waveform with coherent burst-to-burst processing

Figure A.2: Typical Air Surveillance Waveform

PRI seconds. Each sample consists of N_r complex values representing a received pulse; each of these complex values corresponds to a different range cell. The doppler processing stage will calculate an FFT on the values of each sample corresponding to a given range over M_c bursts.

In the subsequent descriptions of the signal processing functions, it will be assumed that data from the sensors is formatted as a large 2-dimensional matrix of size $N_r \times N_{rc}$ representing a single burst. Each function in the pipeline creates a new matrix of complex values; each value is a range-doppler cell.

Pulse Compression

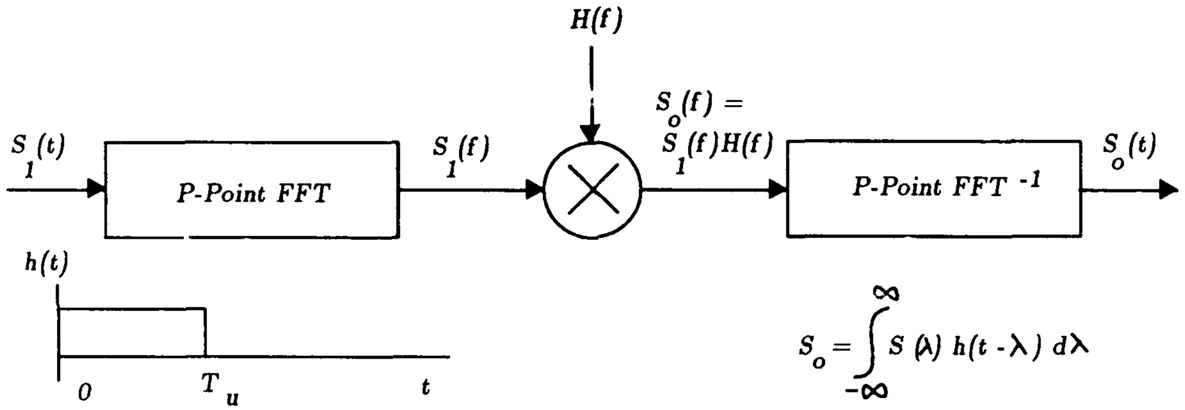


Figure A.3: Digital Pulse Compression

Pulse compression is achieved by taking an FFT over each pulse in each burst, performing a complex multiply by the pulse compression filter frequency transfer characteristic $H(f)$, and taking the inverse FFT of the result. This operation is called a *fast convolver*, and is shown in Figure A.3 [4]. Pulse compression is done for each PRI interval prior to doppler processing. The range cells of each PRI are used in the fast convolver. Therefore, the size of the FFT in the fast convolver is N_r rounded up to the nearest power of two when a radix-2 FFT is used. In terms of the input sensor data matrix, pulse compression is fast convolver on each row of the matrix.

Doppler Filtering

The doppler filter is an FFT over the doppler frequency cells of each group of M_c bursts. Thus, the total FFT size is $M_c \cdot N_{rc}$. This number will be rounded up to the nearest power of two in the actual FFT implementation. In terms of the sensor data matrix, Doppler filtering is an FFT on each column of the large matrix formed out of M_c input matrices collected from the input stream.

Magnitude and Video Integration

Each range-doppler cell is envelope (magnitude) detected and summed (incoherent video integration) over the M incoherent bursts (M different RF frequencies are used during the total dwell time $M \cdot M_c(T_t + T_r)$). The magnitude of the complex value in each range-doppler cell is computed as:

$$|x + jy| = \begin{cases} 0.98|y| + \frac{|x|}{y} & \text{if } y > 2|x| \text{ and } y > x \\ 0.79625|y| + 0.6125|x| & \text{if } y < 2|x| \end{cases}$$

where $x + jy$ denotes the complex value in the cell.

Each dwell (time on target) consists of M coherent processing intervals (CPI) and each CPI consists of M_c bursts. Therefore, each range-doppler cell is incoherently (video) integrated over M samples. Video integration is the summation of the magnitude of each range-doppler cell M times in the total dwell time $M_T(T_t + T_r)$.

CFAR Processing

For constant false alarm processing, the mean level detector (cell averaging) is used. N_c cells at each side of the test cell along the rows of the range-doppler matrix are used to compute the average. If the test cell value is greater than this average, the alarm is sent. If the test cell is not greater than the average, a false alarm is assumed. Figure A.4 shows the mean level detector processing.

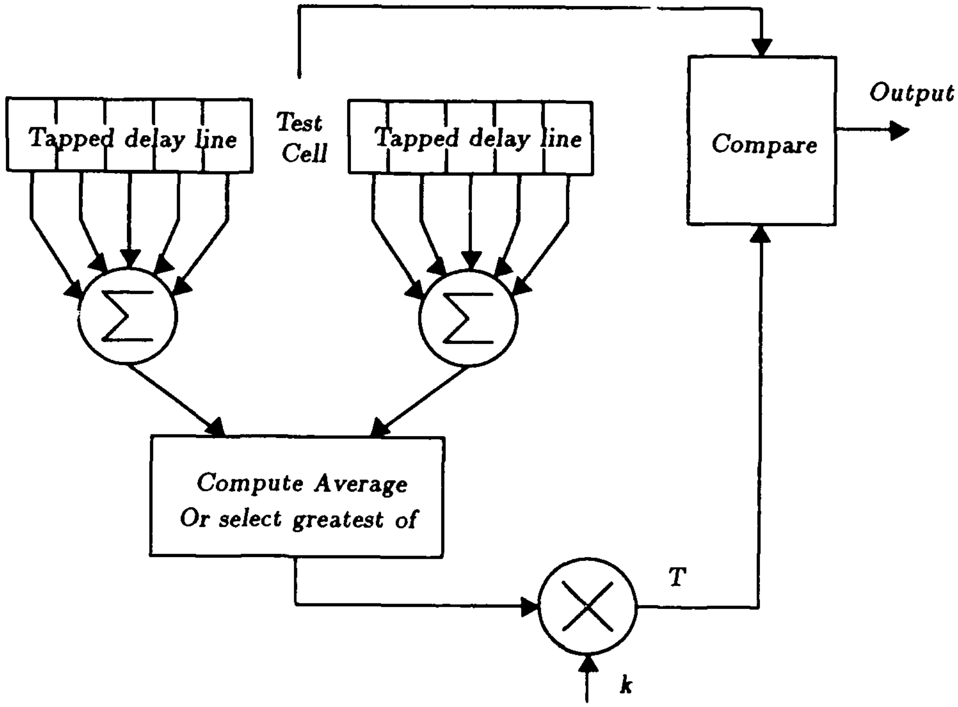


Figure A.4: CFAR Circuit

Appendix B

Application Program Graphs

The following pages contain the program graphs for the functions of the space-based radar application. The code for these functions is presented in Appendix C; the program graphs in this appendix are translated directly from the $\mathcal{PGL}$ code. Operators are inserted in the graph according to the transformation rules presented in Chapter 5.

Figure B.1 shows the program graph for the **radar** function. Inputs are labelled for the purpose of linking with other graphs; this convention is followed in all subsequent graphs. Figure B.2 shows graph with the **Distribute** and **Merge** operators inserted as envisioned by the first-pass optimization process. The connecting arrow between the two operators in the graph denotes that the same values are supplied for the parameters. In general, connections between the parameters of several operators designates that the corresponding parameters are given the same values.

Figure B.3 shows the program graph for the **pulse_compress** function. The constants in the program are denoted by the circled values input to several of the operators, *i.e.*, the constant H is supplied to the v1 input of vect_mult. The operators defined in $\mathcal{PGL}$ are represented as function boxes of the same name. For ease of demonstration, function names are abbreviated in the graph. Figure B.4 shows the program graph for the vect_mult function.

Figure B.5 shows the graph for the pulse_doppler function. The function box **make-matr** denotes the use of an Id array comprehension in the code. This function takes in some number of inputs and combines them into a matrix. Use of array comprehensions is limited in $\mathcal{PGL}$. Each value in the comprehension is assumed to be formed by the same function applied to different values. This allows partitioning over a comprehension as well as an iterator construct.

The constant N which is supplied for the o parameter in these graphs denotes an overlap

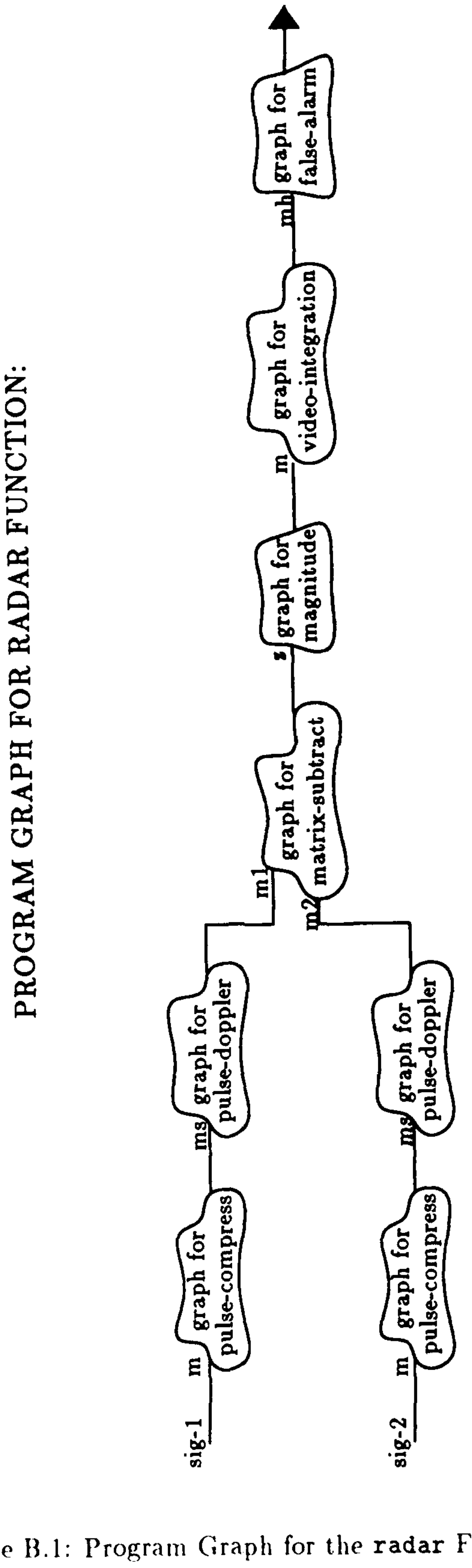


Figure B.1: Program Graph for the radar Function

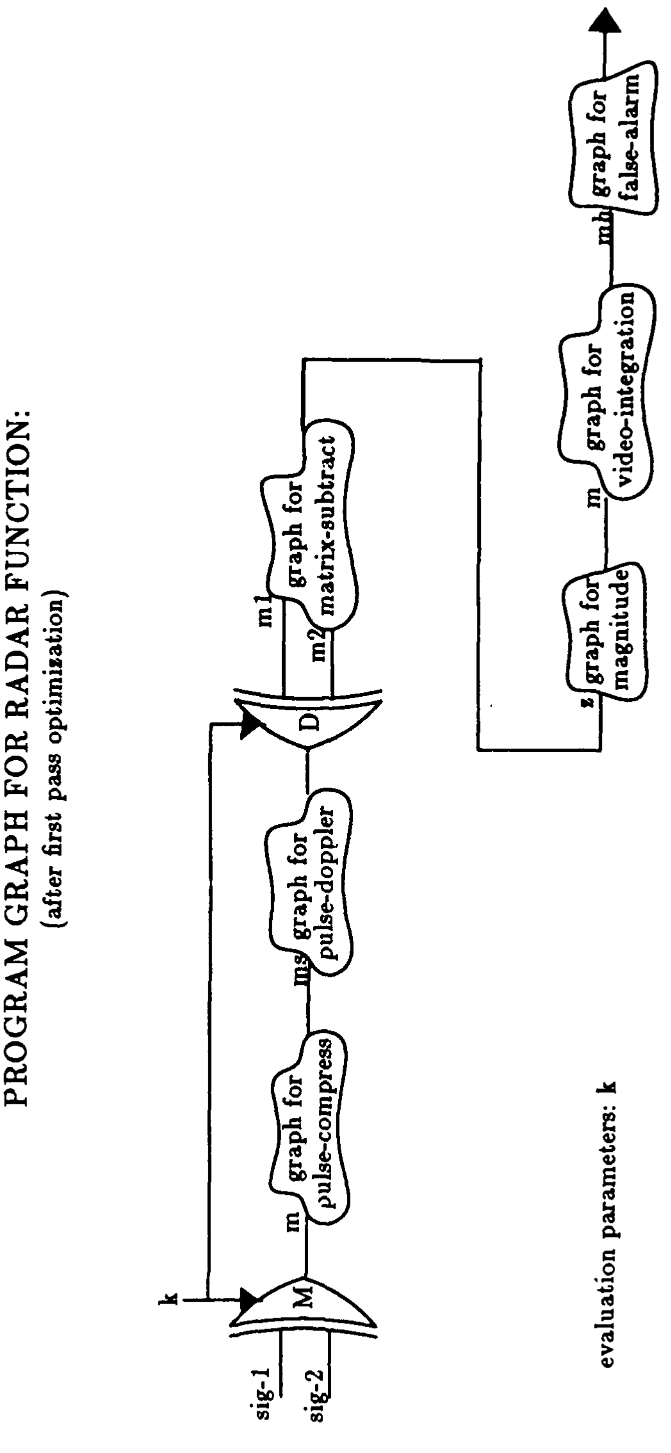


Figure B.2: The **radar** Function after first pass optimization

which depends on the size of the input. If the overlap extends the entire size of the object on the given dimension, then the special value N is used. This enables the same graph to be used on different sizes of inputs.

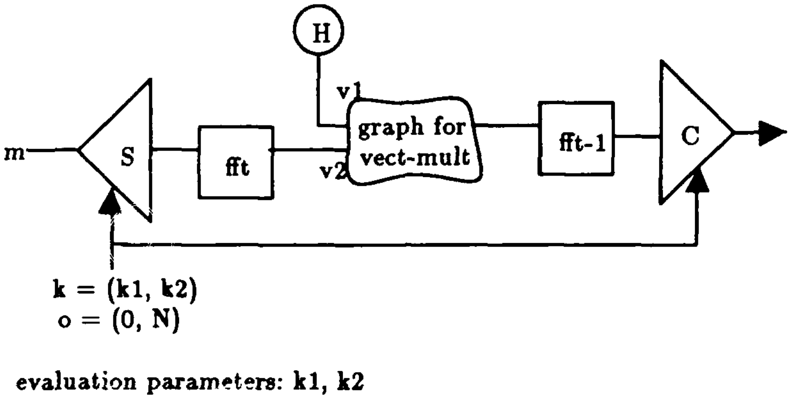


Figure B.3: Program Graph for **pulse_compress** Function

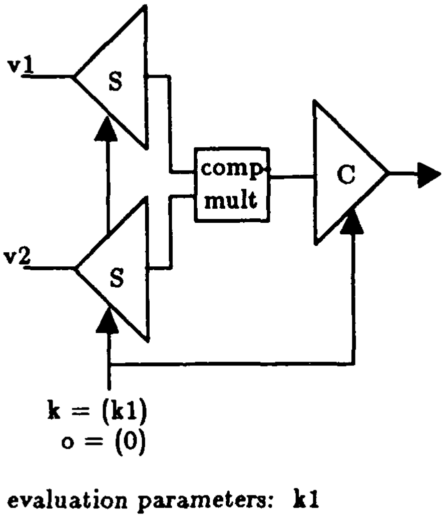


Figure B.4: Program Graph for **vect_mult** Function

Figure B.6 shows the program graph for the **matrix_subtract** function. Figure B.7 shows the graph for the **magnitude** function. Figure B.8 shows the graph for the **video_integration** function. Figure B.9 shows the graph for the **matrix_add** function. Figure B.10 shows the graph for the **false_alarm** function. The false alarm function uses an Id loop construct to define a **map_fold** operation. This is detected by the compiler and transformed appropriately into a special function **map-fold**. Transformation rules for these Id constructs are not specifically

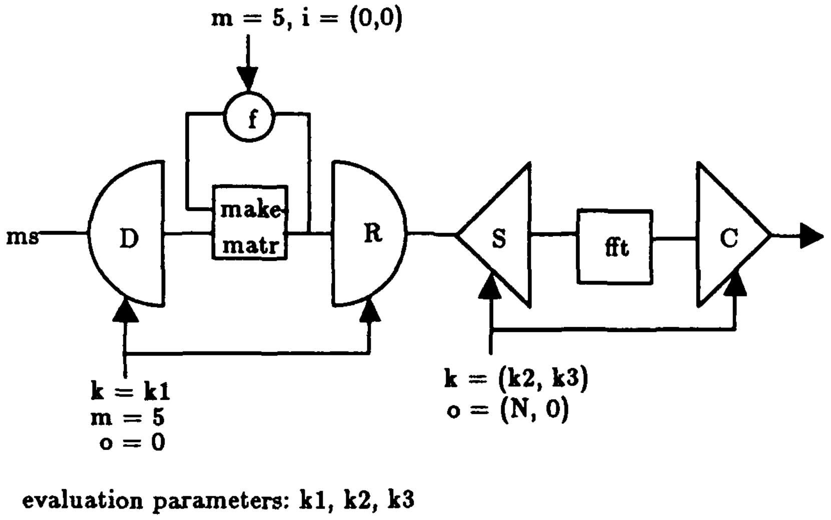


Figure B.5: Program Graph for pulse_doppler Function

defined, however, due to their regular nature, memory and latency costs are derivable. Once these values are derived, they are supplied to the parameters of the function box in the graph. Future versions will eliminate this need.

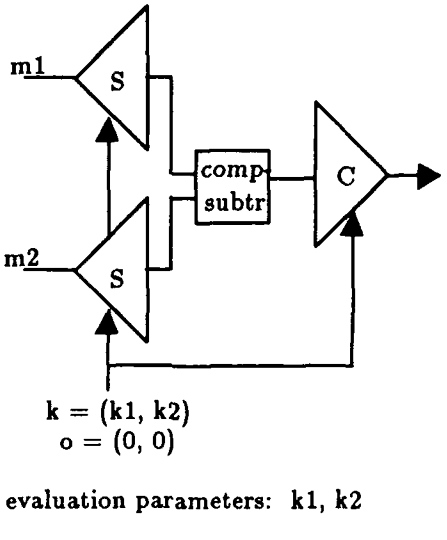


Figure B.6: Program Graph for matrix_subtract Function

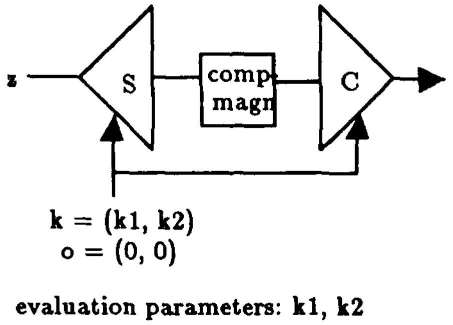


Figure B.7: Program Graph for **magnitude** Function

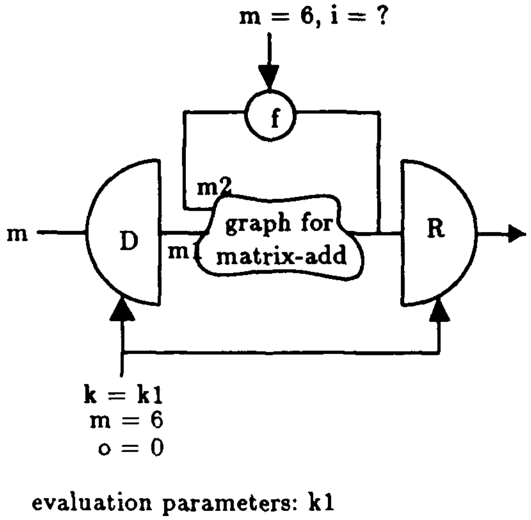
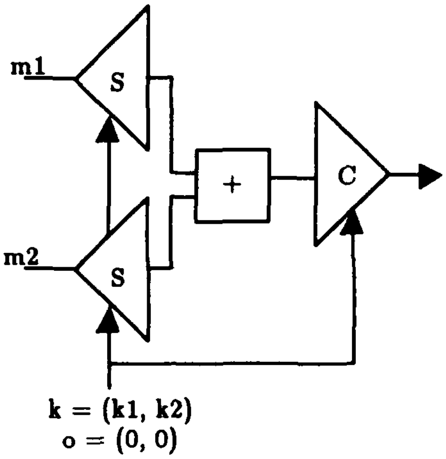
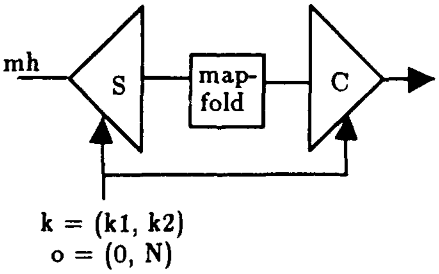


Figure B.8: Program Graph for video_integration Function



evaluation parameters: k1, k2

Figure B.9: Program Graph for `matrix_add` Function



evaluation parameters: k1, k2

Figure B.10: Program Graph for `false_alarm` Function

Appendix C

Application $\mathcal{PGL}$ Code

The following pages contain the $\mathcal{PGL}$ code for the functions of the space-based radar application. The program graphs for these functions are presented in Appendix B; the $\mathcal{PGL}$ programs in this appendix translate directly into the program graphs. Operators are inserted in the graph according to the transformation rules presented in Chapter 5.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% The space-based radar application.
% Defines the processing pipeline and
% data dependencies of functions.
% Top level streams are also defined.
% NR and NRC are globally defined constants.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

def-signal ch_a_sig = make-signal 1 (NR, NRC) 'A;

def-signal ch_b_sig = make-signal 1 (NR, NRC) 'B;

def radar sig_1 sig_2 =

% the signal processing functions.

  {compressed-1 = pulse_compress sig_1;
   compressed-2 = pulse_compress sig_2;

   doppler_1 = pulse_doppler compressed-1;
   doppler_2 = pulse_doppler compressed_2;

   sig_diff = matrix_subtract doppler_1
                              doppler_2;

   sig_magn = magnitude sig_diff;
   sig_video = video_integration sig_magn;
   sig_cfar = false_alarm sig_video
   in
     sig_cfar};
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% The pulse compression function.  For each row of
% the input matrix, perform an FFT, a vector
% multiply and an inverse FFT.  The resulting
% vectors are the rows of the output matrix.  H is
% a globally defined constant.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

def pulse_compress m =

  {generate r from (generate_matrix_rows matrix) in
     fft_1 (vect_mult H (fft r));
   compose compose_matrix_rows};

def vect_mult v1 v2 =

% element-wise complex multiplication of vectors.

  {generate ve1,ve2 from (gen-vector-elements v1),
                         (gen-vector-elements v2) in
     complex-mult ve1 ve2;
   compose comp-vector-elements};
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% The pulse doppler processing.  Collect five
% matrices from the input stream; combine these
% into one large matrix.  FFT is performed over
% each column of the resulting matrix.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

def pulse_doppler ms =

  {new_stream = {init = {matrix ((0,0),(0,0))}
                 in
                   {repeat group over (5-every-5 ms) in
                      combine_into_matrix group init;
                    collect comp-5-5};
   in
     pd new_stream};

def combine_into_matrix m1 m2 =

% combine two matrices into one along the column.
% Assumes number of columns is the same
% (s12 = s22).  All matrices assumed to have
% indeces starting at zero.

  {s11,s12 = 2D_size m1;
   s21,s22 = 2D_size m2;
   in
     {matrix ((0,s11+s21-1),(0,s12-1))
      | [i,j] = m1[i,j] || i <- 0 to s11-1
                         & j <- 0 to s12-1
      | [i,j] = m2[i-s11,j] || i <- s11 to s11+s21-1
                             & j <- 0 to s22-1}};

def pd mat =

% Pulse doppler processing on a large matrix
% consisting of the 5 matrices collected from the
% original input stream.  Performs an fft on each
% column in the matrix.

  {generate c from (gen-matrix-columns mat) in
     fft c;
   compose comp-matrix-columns};
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% The matrix subtraction function.  For each
% element pair, the new element of the output
% matrix is the complex difference.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

def matrix_subtract m1 m2 =

  {generate e1,e2 from (gen-matrix-elements m1),
                       (gen-matrix-elements m2) in
    complex-sub e1 e2;
   compose comp-matrix-elements};

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% The magnitude function.  For each complex
% element, the new element of the output matrix
% is the complex magnitude.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

def magnitude z =

  {generate e from (gen-matrix-elements z) in
     complex-magnitude e;
   compose comp-matrix-elements};
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% The video integration function.  Collect six
% matrices from the input stream, combine them into
% one matrix by summing each pair of elements as
% the matrices arrive.  Uses an Id array
% comprehension to create the initial value.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

def video_integration m =

  {(l0,u0),(l1,u1) = 2D_bounds m;
   init = {matrix (l0,u0),(l1,u1)
           | [i,j] = 0 || i <- l0 to u0
                        & j <- l1 to u1}
   in
     {repeat group over (6-every-6 m) in
        matrix_add group init;
      collect compose-1-6}};

def matrix_add m1 m2 =

% matrix addition.  For each element pair, the new
% element is the sum.

  {generate e1,e2 from (gen-matrix-elements m1),
                       (gen-matrix-elements m2) in
     e1 + e2;
   compose comp-matrix-elements};
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Constant false alarm processing.  Performs
% averaging over the samples in one pulse for
% each pulse (row).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

def false_alarm mh =

  {generate r from (gen-matrix-rows mh) in
     alarm r;
   compose comp-matrix-rows};

def alarm v =

% the actual average and compare operation.  Uses an
% Id loop construct as a map_fold operation.

  {l,u = 1D_bounds v;
   sum = add_elts (select_NC_elts v)
   in
     {for i from l+NC to u do
        next sum = sum - v[i,j-NC] + v[i,j+NC];
        a[i] = (next sum > v[i,j]*2*NC);
      finally a};
```

Bibliography

[1] William B. Ackerman. Data Flow Languages. In *Proceedings of the 1979 National Computer Conference*, pages 1087–1095, 1979.

[2] Arvind and David E. Culler. Dataflow Architectures. Laboratory for Computer Science Technical Memorandum MIT/LCS/TM-294, MIT, 545 Technology Square, Cambridge, MA, February 1986.

[3] R.G. Babb II. Parallel Processing with Large-Grain Data Flow Techniques. *IEEE Computer*, July 1984.

[4] Eli Brookner. Trends in Radar Signal Processing. *Microwave Journal*, 25(10):20–39, October 1982.

[5] Eli Brookner and T.S. Mahoney. Derivation of a Satellite Radar Architecture for Air Surveillance. *Microwave Journal*, 29(2):173–191, February 1986. see also M.I.Skolnik (ed.), Radar Applications. IEE Press, NY, 1988.

[6] Alan Burns, Andrew M. Lister, and Andrew J. Wellings. *A Review of Ada Tasking (Lecture Notes in Computer Science; 262)*. Springer-Verlag, Berlin, 1987.

[7] M.L. Campbell. Static Allocation for a Data Flow Multiprocessor. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 511–516. IEEE Computer Society Press, 1985.

[8] Marina C. Chen. Very High-level Parallel Programming in Crystal. Research Report YALEU/DCS/RR-506, Department of Computer Science, Yale University, New-Haven, CT, December 1986.

[9] Daniel J. Dechant and Frank A. Horrigan. AOSP - Multiprocessor Architecture and System Considerations. In *AIAA Computers in Aerospace VI*, Wakefield, MA, October 1987.

[10] C. Gao, J.W.S. Liu, and M. Railey. Load Balancing Algorithms in Homogeneous Distributed Systems. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 302–6. IEEE Computer Society Press, 1984.

[11] T. Gautier, P. Le Guernic, and L. Besnard. SIGNAL: A Declarative Language for Synchronous Programming of Real-Time Systems. In *Functional Programming Languages and Computer Architectures (Lecture Notes in Computer Science; 274)*. Springer-Verlag, 1987.

[12] E. Gehringer, D.P. Siewiorek, and Z. Segall. *Parallel Processing: the CM* Experience*. Digital Press, Cambridge, MA, 1987.

[13] James E. Hicks, Jr. A High-level Signal Processing Programming Language. Master's thesis, MIT, Cambridge, MA, 1988.

[14] L.Y. Ho and K.B. Irani. An Algorithm for Processor Allocation in a Dataflow Multiprocessing Environment. Technical report, IEEE, 1983.

[15] A.J. Jagodnik, J.R. Samson, Jr., and M.J. Young. A Multifaceted Architecture for Signal Processing. Internal memorandum, Raytheon Company, Sudbury, MA, 1983.

[16] Douglas Johnson et al. Automatic Partitioning of Programs in Multiprocessor Systems. Technical report, IEEE, 1980.

[17] E.J Kelly, G.N. Tsandoulas, and V. Vitto. A Displaced Phase Center Antenna Concept for Space Based Radar Applications. In *Military Microwaves Conference Proceedings*, pages 154–164, London, October 1984.

[18] Edward A. Lee and David G. Messerschmitt. Pipeline Interleaved Programmable DSP's: Synchronous Data Flow Programming. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(9), September 1987.

[19] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, MA, 1986.

[20] J.R. McGraw and S.K. Skedzielewski. Streams and Iteration in VAL: Additions to a Data Flow Language. Technical report, IEEE, 1982.

[21] R. Mehrotra and S.N. Talukdar. Scheduling of Tasks for Distributed Processors. Technical report, IEEE, 1984.

[22] R.S. Nikhil. Id Reference Manual. Computation Structures Group Memo 284, Laboratory for Computer Science, MIT, 545 Technology Square, Cambridge, MA, March 1988.

[23] D.A. Padue and M.J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[24] D.A. Reed, L.M. Adams, and M.L. Patrick. Stencils and Problem Partitionings: Their Influence on the Performance of Multiple Processor Systems. *IEEE Transactions on Computers*, 36(7):845–58, July 1987.

[25] Guy L. Steele, Jr. *Common Lisp.* Digital Press, Cambridge, MA, 1984.

[26] W. Wadge and E. Ashcroft. *LUCID: The Dataflow Programming Language.* Academic Press, Inc., London, 1985.

[27] Elizabeth Williams. Assigning Processes to Processors in Distributed Systems. Technical report, IEEE, 1983.

OFFICIAL DISTRIBUTION LIST

Director 2 copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Office of Naval Research 2 copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

Director, Code 2627 6 copies
Naval Research Laboratory
Washington, DC 20375

Defense Technical Information Center 12 copies
Cameron Station
Alexandria, VA 22314

National Science Foundation 2 copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

Dr. E.B. Royce, Code 38 1 copy
Head, Research Department
Naval Weapons Center
China Lake, CA 93555